平成２５年度文部科学省委託
東日本大震災からの復旧・復興を担う専門人材育成支援事業

実践！自動車組込み技術者講座

ＦＰＧＡとマイコンの連携システム

（ソフトウェア応用編）

~The first step is the only difficulty.

東北の復興を担う自動車組込みエンジニア育成支援プロジェクト推進協議会

Content

0 はじめに 1

1 組込み及び車載システムの概略 2

1.1 組込みシステムとは 2
1.2 組込みシステムの特性 3
1.3 自動車に見る組込みシステムの傾向 5
1.4 車載ネットワークの概要 7
1.4.1 CAN 7
1.4.2 LIN 7
1.4.3 MOST 8
1.4.4 1394-Automotive 8
1.4.5 FlexRay 8

2 リアルタイム OS の必要性 9

2.1 OS とは 9
2.2 OS搭載時の問題点 9
2.3 リアルタイム OS の概要 10
2.4 並行処理単位とは 13
2.5 リアルタイム OS を使用するメリット 14
2.5.1 ハードウェアの抽象化 14
2.5.2 プログラム開発の分業化 14
2.5.3 実行時間管理 15
2.6 リアルタイム OS を使用するデメリット 17
2.6.1 速度的なオーバヘッド 17
2.6.2 スタック領域の増加 17
2.6.3 排他制御が必要 18
2.7 割込み処理 19
2.7.1 割込みとは 19
2.7.2 割込みの仕組み 20
2.7.3 OS における割込み処理 21

3 OSEK/VDX 仕様概論 23

3.1 OSEK/VDX とは 23
3.2 OSEK/VDX の特徴 23
3.3 OSEK OS の特徴 24
3.4 OSEK OS の仕組み 24
3.4.1 スケジューリング方法の違い 24
3.4.2 コンフォーマンスクラス 29
3.4.3 スケジューリング方式 31
3.4.4 イベント 32
3.4.5 アラーム 33
3.4.6 リソース 34
3.4.7 フックルーチン 35

4 リアルタイム OS を使用した組込み開発手法 36

4.1 組込みシステム開発 36
4.1.1 組込みシステム開発の流れ 36
4.1.2 クロス開発環境 36
4.1.3 ROM 化 38
4.2 リアルタイム OS の有無による開発の違い 39
4.2.1 リアルタイム OS を使用しない場合の開発 39
4.2.2 リアルタイム OS を使用した場合の開発 41
4.2.3 コンフィギュレーション 43
4.3 デバッグ手法 43
4.3.1 シミュレータ 43

5 TOPPERS Automotive Kernel の使用方法 45

5.1 ファイルの種類とディレクトリ構成 46
5.1.1 ディレクトリ構成 46
5.1.2 カーネル 46
5.1.3 アプリケーション 47
5.1.4 システムジェネレータ 47
5.2 プログラム作成手順 47
5.2.1 プロジェクトの新規作成 47
5.2.2 プロジェクトの各種設定 54
5.2.3 プロジェクトへのファイル登録 59
5.2.4 システムジェネレータの使用方法 62
5.2.5 プロジェクトへのシステムジェネレータ登録 63
5.2.6 OIL ファイルの作成 66
5.2.7 アプリケーションの作成 69
5.3 プログラムの書込み 74
5.3.1 書込み・デバッグ準備 74
5.3.2 書込み手順 75
5.3.3 デバッグ手順 79

6 マルチタスクプログラミング 84

6.1 タスクの作成 84
6.2 タスク制御 84
6.3 アラーム機能 94
6.4 排他制御 97
6.4.1 優先度上限プロトコル 97
6.4.2 リソースの使用方法 98
6.4.3 リソース機能使用時の注意 99
6.4.4 常に排他制御を意識する 99

7 MISRA-C 101

7.1 コーディング規約とは 101
7.1.1 C 言語コーディング規約 101
7.1.2 C 言語にコーディング規約が必要な理由 102

7.2　MISRA-C とは 102
7.2.1　MISRA-C の背景 103
7.2.2　MISRA-C の効果 103
7.2.3　MISRA-C の特徴 103
7.2.4　MISRA-C ルールの例 105
7.3　MISRA-C 実施プロセス 108
7.3.1　合致マトリクス 109
7.3.2　MISRA-C チェック方法 110
7.3.3　エラー対応方法 111
7.3.4　逸脱手続き方法 111

8　デバイスドライバ 114

8.1　デバイスドライバとは 114
8.1.1　デバイスドライバの役割 114
8.1.2　デバイスドライバの構造 115
8.2　デバイスドライバを使う 116
8.2.1　提供デバイスドライバについて 116
8.2.2　SFR 140
8.3　提供デバイスドライバを用いたアプリケーションの作成 141
8.3.1　システム概要 141
8.3.2　ソフトウェア仕様 142

9　CAN 通信 144

9.1　CAN 通信プロトコル 144
9.1.1　CAN コントローラ 144
9.1.2　コントローラフォーマット 144
9.1.3　通信バス規格 145
9.1.4　ドミナントとリセッシブ 145
9.1.5　同期とスタッフィングルール 146
9.1.6　マルチマスタ方式 / イベントドリブン方式 146
9.1.7　アービトレーション（調停） 147
9.1.8　フレーム 148
9.1.9　エラー状態 153
9.1.10　ビットタイミング 154
9.1.11　アクセプタンスフィルタ 154
9.2　CAN 通信ドライバの開発 155
9.2.1　CAN デバイスドライバ構成 155
9.2.2　製作の環境準備 157
9.2.3　コールバック 157
9.2.4　依存部関数 159
9.2.5　API 167
9.2.6　コールバック関数 171
9.2.7　OIL ファイル（ISR） 172
9.3　CAN デバイスドライバを用いたアプリケーションの作成 174
9.3.1　システム概要 174
9.3.2　ソフトウェア仕様 174
9.3.3　TOPPERS Platform ボード間の接続方法 180

10 応用アプリケーションの開発 185

10.1 概要 185
10.1.1 システム概要 185
10.1.2 システム仕様 186
10.1.3 CAN 通信仕様 186
10.2 ハードウェア環境 187
10.2.1 TOPPERS Automotive Kernel 実装環境 187
10.2.2 搭載MCUのCANモジュール仕様 190
10.3 簡易距離計測システムの動作 193
10.4 OBD—Ⅱ 194

11 演習問題解答例 197

11.1 6.3 章 - アラーム 197
11.1.1 OIL ファイル 197
11.1.2 ヘッダファイル 198
11.1.3 ソースファイル 198
11.1.4 プログラムの動作 200
11.2 6.4.4 章 - リソース 201
11.2.1 OIL ファイル 201
11.2.2 ヘッダファイル 202
11.2.3 ソースファイル 202
11.2.4 プログラムの動作 204
11.3 7.3 章 –演習問題 1 207
11.4 7.3 章 - 演習問題 2 207
11.5 8.2.8 章 - シリアル通信プログラムの作成 208
11.5.1 OIL ファイル 208
11.5.2 ヘッダファイル 208
11.5.3 ソースファイル 209
11.6 8.4 章 - 提供デバイスドライバを用いたアプリケーションの作成 212
11.6.1 OIL ファイル 212
11.6.2 ヘッダファイル 213
11.6.3 ソースファイル 213
11.7 9.1 章 - CAN 通信プロトコル 217
11.8 9.2 章 - CAN 通信ドライバの開発 217
11.8.1 ヘッダファイル 217
11.8.2 ソースファイル 218
11.8.3 依存部ヘッダファイル 220
11.8.4 依存部ソースファイル 222
11.9 9.3 章 - CAN デバイスドライバを用いたアプリケーションの作成 228
11.9.1 （送信アプリ）OIL ファイル 228
11.9.2 （送信アプリ）ヘッダファイル 230
11.9.3 （送信アプリ）ソースファイル 230
11.9.4 （受信アプリ）OIL ファイル 233
11.9.5 （受信アプリ）ヘッダファイル 235
11.9.6 （受信アプリ）ソースファイル 235

12 Appendix B （MISRA-C ルール一覧） 239

はじめに

現在の私たちの生活基盤を支えている「自動車」は、今や基本性能だけにとどまらず、安全性能、環境性能、快適性能といった、さまざまな要求を満たさなければなりません。そのため、ECU（Electric Control Unit；電子制御装置）と呼ばれる非常に多くのコンピュータを搭載し、ネットワークにより協調制御することで実現しています。しかし、システムの爆発的な大規模化が問題となっています。

この問題に対応するためには、「開発作業の効率化」や、「資産流用」が必須であり、これらを実現するために、組込み OS（Operating System）が採用されています。現状、多くの ECU は OS を搭載していませんが、一部の ECU には OSEK/VDX という車載における「デファクトスタンダード（事実上の標準）」である OS が搭載されているものもあります。また、今後の共通プラットフォーム化に伴い、OS の搭載は加速するものと思われます。

また車載ネットワークでは、CAN（Controller Area Network）、LIN（Local Interconnect Network）といった通信プロトコルが標準的に使われており、車載システムを開発する上で欠かせない技術となっています。

本書は、車載システムを中心とした組込みシステム全般に必要な技術である、OSEK/VDX OS、デバイスドライバ、MISRA-C、CAN などについて詳細に説明しています。OS には、TOPPERS Automotive Kernel(旧称 TOPPERS/OSEK カーネル)を用いて、実際に現場で使用できる「実践的な知識の習得」を目標にしています。また、CAN、LIN デバイスドライバを開発し、実際に使用することで、車載ネットワークに関する技術の習得も行います。

本書は、CAN、LIN、FlexRay といった車載ネットワークを搭載した「TOPPERS Platform ボード」を用いて、下記の手順で説明していきます。

- TOPPERS Automotive Kernel の使用
- TOPPERS Automotive Kernel 上で動作するアプリケーションの作成
- デバイスドライバの使用
- CAN デバイスドライバの作成
- 応用アプリケーション（CAN 通信ゲートウェイ）の作成

本書は、「組込みシステムとは」という初心者向けの内容から「CAN/LIN 通信ゲートウェイの作成」という中級者向けの内容まで幅広く説明しています。

本書が、高い技術を持った組込み技術者の教育、さらに組込みシステムに興味を持つ方々が増えることにお役立てできれば幸いです。

本書を作成するために、ご多忙にもかかわらず多方面からご協力をいただきました。ご協力をいただいた方々にこの場を借りて御礼申し上げます。

1 組込み及び車載システムの概略

1.1 組込みシステムとは

私たちが日ごろ生活をする上で必要な機械・道具にはどのようなものがあるでしょうか？

たとえば、ごはんを作るのに必要な炊飯器、冷蔵庫、電子レンジ、通勤通学で利用するバス、電車、自動車や情報・娯楽のテレビ、携帯型ステレオ、携帯電話など多くの便利な道具を使っています。これらの中にはごくまれに機械部品だけで構成されるもの（例：機械式の腕時計）もありますが、ほとんどの製品は一種のコンピュータを内蔵しています。このコンピュータのことを「マイクロコンピュータ（通称：マイコン）」といいます。

一般的なマイコンには、中央演算処理装置（CPU、Central Processing Unit）、割込みコントローラ、内蔵 ROM（Read Only Memory）、内蔵 RAM（Random Access Memory）、汎用入出力装置（デジタル I/O ポート、A/D コンバータ）、汎用デバイス（タイマ、PWM：Pulse Width Modulation）コントローラ、WD (Watchdog timer)、内蔵クロックユニットなどを含んでいますが、“マイクロ”というだけあってコンピュータの規模としては非常に小さくまとめられています。一方、汎用コンピュータは CPU やメモリは個別に用意されており、それぞれの規模や能力は比較にならないほど大きなものです。

利用者の利便性や製品としての価値を実現するために、マイコンを利用して求められる要求事項を満たしている製品を「組込み機器製品」といい、これらの制御システムを「組込みシステム」といいます。さらに、組込みシステムは、組込み制御装置（マイコン及び機器制御用ハードウェア）と組込みソフトウェアで構成されています。

すなわち「組込み機器」= 製品専用機械部品 + 「組込みシステム =（組込み制御装置 + 組込みソフトウェア）」となります。本書では、「組込みソフトウェア」について重点的に説明します。

＜組込みシステムの定義＞

組込みシステムの定義には幅広いものがあります。なぜなら組込みシステムの応用は多様（炊飯器から電子力発電所の制御まで）であり、その性質の違いから明確に定義できないといわれているからです。しかしその中で、以下に示すようなすべての組込みシステムで共通に利用できる特徴を、組込みシステムとして定義付けることができます。

組込みシステムとは、“各種の機械・機器に組み込まれて、その制御を行うコンピュータシステム”。

すなわち、ソフトウェアとハードウェアが一体・不可分なものとして人に認識されるものであるといえます。

一方、組込みシステムと対比する用語として“汎用システム”という言葉があります。汎用システムはいろいろな目的を実現できる可能性を持ったハードウェアと、目的を実現するためのソフトウェア（ソフトウェアは目的達成のために作られる場合が多い）で構成されますが、ソフトウェアは一種類ではなくいろいろなソフトウェアが利用されます。汎用システムでもっとも身近に存在するものは、パーソナルコンピュータ（パソコン）です。汎用性の高いハードウェア上にいくつかの専用ソフトウェア（ワープロソフト、表計算ソフト、ゲームソフトなど）が動作し、同じ製品を複数の目的で利用できるのが汎用システムです。

～コラム～ パソコンは組込みシステムともいえる？？
一般的な説明では、パソコンは汎用システムです。しかしパソコン“だけ”を取り上げると、パソコンも立派な組込みシステムです。パソコンにはプロセッサやメモリなどいくつかの装置が搭載されており、それら装置を制御するソフトウェアを“BIOS（Basic Input / Output System）”といいます。この BIOS は“その”パソコンに搭載されている装置を扱うソフトウェアであり、他所から勝手に装置を取り入れても動作しません。すなわち、BIOS は BIOS を搭載しているパソコン上でのみ動作し、当該パソコンを汎用 OS などが利用できるようにするためのものです。従って、パソコン装置と BIOS の組み合わせであれば、立派な組込みシステムといえます。

＜組込みシステムの例＞
組込みシステムの応用は広く、多様性があると説明しました。それでは実際にどのような製品に使われているかの例を示します。

- -**家電機器** （電子レンジ、炊飯器、冷蔵庫、洗濯機、乾燥機、エアコン･･)
- -**AV機器**（テレビ、ビデオ（DVD）、デジタルカメラ、オーディオ機器 ･･･)
- -**娯楽・教育機器**（ゲーム機、電子楽器、カラオケ、パチンコ･･･)
- -**個人用情報機器** (PDA、電子手帳、カーナビ･･･)
- -**パソコン周辺機器**（プリンタ、スキャナ、ディスク、CD-ROMドライブ･･･）
- -**OA機器**（コピー、FAX ･･･)
- -**通信機器 端末** (電話機、留守番電話機、携帯電話機 ･･･)
- -**ネットワーク設備**（交換機、PBX、ネットワークルータ、ハブ･･･）
- -**運輸機器**（自動車、信号機、鉄道車両/制御、航空機、船舶･･･）
- -**工業制御・FA機器**（プラント制御、工作機械、工業用ロボット･･･）
- -**設備機器**（ビル用照明、ビル用空調、ビル用電力システム、エレベータ･･･）
- -**医用機器・福祉機器**（血圧計、心電計、レントゲン、CTスキャナ ･･･）
- -**宇宙・軍事**（ロケット、人工衛星、ミサイル ･･･）
- -**その他の業務用機器**（業務用データ端末、POS端末、自動販売機 ･･･）
- -**その他の計測機器**（シンクロスコープ、ICテスタ、電子メータ ･･･）

なお、本書は車載向けの組込みソフトウェアを主な対象とするため、上記運輸機器に利用されるシステムを中心に説明します。

1.2 組込みシステムの特性

組込みシステムは、システム全体で 1 つの目的（携帯電話などでは複数の目的といえるかもしれません）を達成するために開発されたものであるため、いくつかの特性があります。以下に示す 4 つの特性は、特に重要といわれる特性です。

1. 専用化されたシステム
システム全体が 1 つの目的に専用化して設計されています。ハードウェアもソフトウェアも専用化されています。そのため他の目的を考慮する必要がなく、要求される使われ方に限定して、コストを抑えた開発や生産が可能となります。一方で例え同じメーカの同じ製品シリーズであっても、異なる動作や要求があるため、それぞれのシリーズで専用設計しなければなりません（同じシリーズであっても同じソフトウェアを動作させることはできません。当然、ソフトウェアの流用は可能です）。

2. 厳しいリソース制約

年間多くの組込みソフトウェアが開発されますが、その多くが大量生産品向けに開発されます。大量生産品は、生産コスト（生産するために必要な費用で工場組立費や材料代を示します）を抑えることにより、生産コスト・部材コストが利益に直結します（1,000,000 個生産する製品で 1 個あたり 100 円のコスト削減を実現したら 100,000,000 円の削減が可能ですが、10 個しか生産しない製品では 1,000 円しか削減されません。そのため、大量生産品は生産コストを削減することが重要です）。そのため、組込みシステムの多くは少しでも安いマイコン、少ない ROM/RAM での実現が求められます。

一方、開発費は開発したときに一度だけ必要となる費用です。仮に、安いマイコンを利用するために開発費が増加したとしても、生産コストが抑えられれば、企業は利益を得られます。そのため、組込みシステム開発現場では、生産コストを低減するために最適化設計に注力する場合も多く見られます。

さらに、低消費電力、動作環境（温度条件）、軽量化などのリソース制約が求められます。携帯電話を例にしても、待ち受け時間を長くしなければなりませんし、自動車の場合でも、エンジンが停止しているときに動作しなければならないシステム[1]も多く存在します。

また、自動車は北極圏でも赤道直下でも利用できなくてはいけません。自動車のコンピュータは-30 度以下から 100 度ぐらいまでの温度範囲で動作する能力が必要です。さらにシステム重量は燃費を左右するため、いかに軽量化を進めるかも重要な技術となります。

これらのリソース制約は汎用システムに比べ組込みシステムの重要な特性といえます。

3. 高い信頼性

組込みシステムは他の汎用システムと比較して高い信頼性を要求されます。当然のことですが汎用システムも信頼性は要求されます。しかし、利用される製品や性質により、一般的に汎用システムより組込みシステムはより高い信頼性が必要となります。

いくつか例を示します。毎朝みなさんはご飯を食べると思います。仮に、朝起きたとき、炊飯器の不具合でご飯が炊けていなかったらがっかりするのではないでしょうか？

このように、組込みシステムはいついかなるときでも期待したサービスは提供される“もの”として考えられています。いわゆる、当たり前の信頼性です。

自動車などのシステムを例にすると、ハンドルのパワーステアリングやブレーキは、運転中に動作しなくなるなど考えられません。仮に、動作しなくなると事故を起こす可能性があります。このように組込みシステムはシステムの問題により事故など人命を危険にさらす可能性を含んでいるため、絶対的な信頼性を必要とするシステムに使われています。

一方、汎用システムはどうでしょうか？確かに汎用システムの不具合で何時間も作業をしたにもかかわらず、データを紛失することは問題でしょう。しかし、直接的に人命を危険にさらすことはほとんどありません。問題が発生すればリセットできる汎用システムと比べ、リセットすることができない組込みシステムは高い信頼性を必要とします。

4. リアルタイム性

組込みシステムの多くはリアルタイム性を必要とします。このリアルタイム性とは、単に応答が速ければよいということでなく、サービスの要求からサービス開始までの時間が予測可能であることが重要です。みなさんが良く利用している Microsoft Windows シリーズはリアルタイム OS とはいえません。マルチタスクマルチユーザの OS ではありますが、リアルタイム性のある優先度ベースのスケジューリングとはいえないものです。

最近のパソコンは高性能になっているため、ボタンを押すなどのユーザ要求にある程度の速度で応答しますが、いつでも同じ時間で処理を実行もしくは処理が完了するとはいえません。このような性質の OS もしくはシステム部品で、自動車のブレーキシステムを構築したらどうなるでしょう？ブレーキを掛けたいときに、大きな処理を実行しているからブレーキ処理が遅れるとなると、大事故につながりかねません。このようなシステムでは、ブレーキの応答時間（処理要求から動作開始）は X msec 以内という条件が要求事項となります。そのため、この例のような組込みシステムは要求時間を確実に達成する能力を有する性能が必須です。

[1] 代表例としてドア・ロックシステムがあります。ドア・ロックはエンジンが停止しているときに、利用者が所持するキースイッチの押下により反応する必要があります。これにはエンジンが停止していてもマイコンは動作し続ける必要があります。すなわち、自動車のバッテリを利用して動作しているのですが、長時間（数週間以上）乗らない場合でもバッテリ上がりしてはいけません。これらのシステムは低消費電力技術を駆使して実現しています。

1.3 自動車に見る組込みシステムの傾向

現在の傾向を、組込みシステムの代表格である「自動車」を例に説明します。

近年の自動車は、“基本性能”“安全性能”“快適性能”“環境性能”など、顧客や社会から要望される性能が急速に高まっています。この要求を満たすために、組込みシステム技術を利用し対応しています。具体的には、ECUといわれるコンピュータを数多く[2]利用して実現しています。

図 1.3-1 ECU 利用機能例

ECU の制御の例としては、センサから読み取ったデータを元に制御量を決定してアクチュエータを制御する、といったものがあげられます。

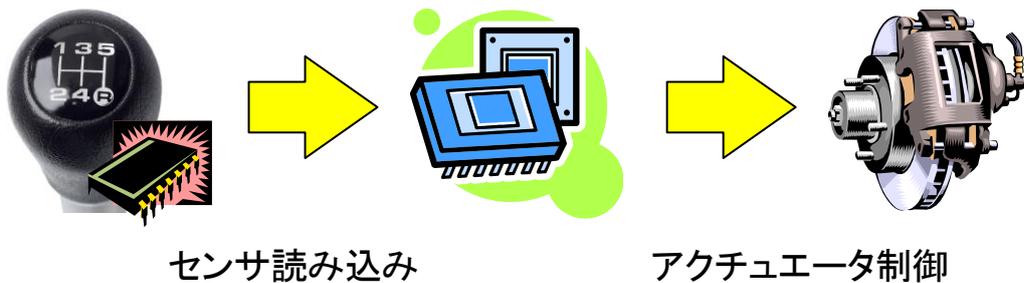



図 1.3-2 ECU 制御例

自動車への要求事項は増加の傾向を辿っており、その結果 ECU 搭載可能数は限界[3]に達しつつあります。ソフトウェア開発量も爆発的に増加し、開発技術者への負担の増大や、品質に支障をきたす事態になりつつあります。この ECU 搭載個数及びソフトウェア開発量の増加が自動車制御システムの大きな課題となっています。

この課題を軽減・改善するための方法として、基盤ソフトウェアの標準化、ソフトウェアフレームワーク、ECU 統合などの方法が模索されています。

[2] 近年の高級車の場合、ECU は 100 個程度搭載されているといわれています。

[3] 自動車にはコンピュータを配置できる環境条件を有する箇所は少なく、多くの ECU は助手席の足元に搭載することになります。しかしこの場所に多くの ECU を搭載することは不可能であり、100 個を超える ECU を搭載するのは空間的に困難となっています。

基盤ソフトウェアの標準化

ISO（International Organization for Standardization）や IEC（International Electrotechnical Commission）などの規格の国際標準化は欧州が中心[4]となっています。自動車分野においても自動車電子技術の標準化は欧州が中心となり、過去では OSEK/VDX[5] , HIS[6] などの標準化団体が活躍していました。これらの団体の成果を有効に活用し、現在では AUTOSAR（Automotive Open System Architecture ; http://www.autosar.org/）を中心とした自動車電子技術の標準化活動が国際的に活発になっています。

一方、日本国内においても JasPar（Japan Automotive Software Platform Architecture ; https://www.jaspar.jp/）と呼ばれる国内の自動車電子技術標準化団体が設立され、国内の標準化活動が進められています。

これらの標準化の目的は、基盤ソフトウェア部位（OS や各種ミドルウェア）の標準化が進むことにより、ソフトウェアの流用性を確保し、ソフトウェア単体での流通を促進することです。また、JasPar ではこれらの基盤ソフトウェアはメーカ間で競争をするものではなく、共通で良いものを作成するべきものであると考えており、この分野での基盤ソフトウェアの標準化は今後一層進むものと考えられます。

ソフトウェアフレームワーク

前述した AUTOSAR は、ソフトウェアのフレームワークも規定しています。BSW（Basic Software）と呼ばれる基盤ソフトウェア部位は複数のコンポーネントで構成されており、それぞれのコンポーネント間やコンポーネントとアプリケーション間を RTE（Run-Time Environment）と呼ばれる自動生成コードで結合します。このようなフレームワークを規定することにより、部品流通を促進するばかりでなく、ソフトウェア部品の組み合わせによる製品の開発を目標としています。

一方、現状の国内自動車ソフトウェアにおいても複数の既存コードは流用されていますが、結合及び組み合わせに関する項目は、詳細なテストを実施して安全を確保しています。このようなフレームワークを利用することにより、接続される部品単体での品質と呼び出され時の動作などが規定され、結合・組み合わせ時に発生する問題を最小限に抑え、詳細なテストを省略することを意図しています。しかし、実際に利用できる品質と安全を維持できるのかという点には疑問が残ります。

ECU 統合

近年の自動車は求められる顧客要求に応えるために ECU を利用して要求を実現し、結果として ECU 数が爆発的に増え、その搭載限界を超えていることは既に説明しました。しかし、今後も自動車に求められる要求事項を実現するためには ECU 利用は避けられず、ECU 増加は進むと考えられます。

そのため、自動車メーカによっては複数の ECU を 1 つに統合する ECU 統合により個数削減を実現[7]しつつあります。このような ECU 統合に有益な技術として、基盤ソフトウェア（一般的にはリアルタイム OS）の保護機能[8]があげられます。保護機能とは、メモリ保護や時間保護と言った保護機能を有し、異なるメーカで開発されたソフトウェアが他のソフトウェアに影響を与えないようにする技術です。この技術を有益に使うことにより、それぞれのソフトウェアはそれぞれ別の ECU 上で動作する場合と同じ条件で開発することができます。

自動車制御における組込みシステム及びソフトウェアの動向は、国内の組込みシステムの動向を牽引しています。技術の発展は、活発な活動をしている業界が引っ張るものであり、ここ数年自動車関連の組込み技術が大きく他をリードしています。また自動車で培われた組込み技術は、ロボット、産業機械など幅広く利用できる技術です。なぜなら自動車は人を乗せる道具であり、人を含めた地域社会、安全、品質を考える必要があります。今後、組込み技術を応用した製品は、ますます人と密接（介護ロボットなどはその代表例）になります。そのため、人を含めた「安全・安心」技術は自動車の技術から進化していくでしょう。

---

[4] 標準化は投票により決定します。この投票権は国毎にあるため、欧州は小さな国が集まっているため投票数が有利です。そのため国際標準化は欧州中心となっています。

[5] OSEK = 独：Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug、英語：Open system together with interfaces for automotive electronics　VDX = Vehicle Distributed eXecutive　(http://www.osek-vdx.org/)

[6] Herstellerinitiative Software　(http://www.automotive-his.de/)

[7] トヨタ自動車から発売されているレクサス LS は異なる部品メーカ 3 社の ECU を 1 つに統合しています。

[8] 保護機能をつけたリアルタイム OS として、株式会社ヴィッツが開発した保護 OS があります。http://www.witz-inc.co.jp/pi/ipaward/IPaward.html

1.4 車載ネットワークの概要

各機能に求められる通信品質の高さとコストの兼ね合いにより、各 ECU 間で使われるネットワークプロトコルにはさまざまな種類のものが存在します。各社独自の通信プロトコルも存在しますが、ここでは一般的なもののみ紹介します。

※参照：EE Times Japan「クルマに広がるネットワーク（前編）　電子化の背景と車載 LAN の種類」
http://eetimes.jp/article/20492/

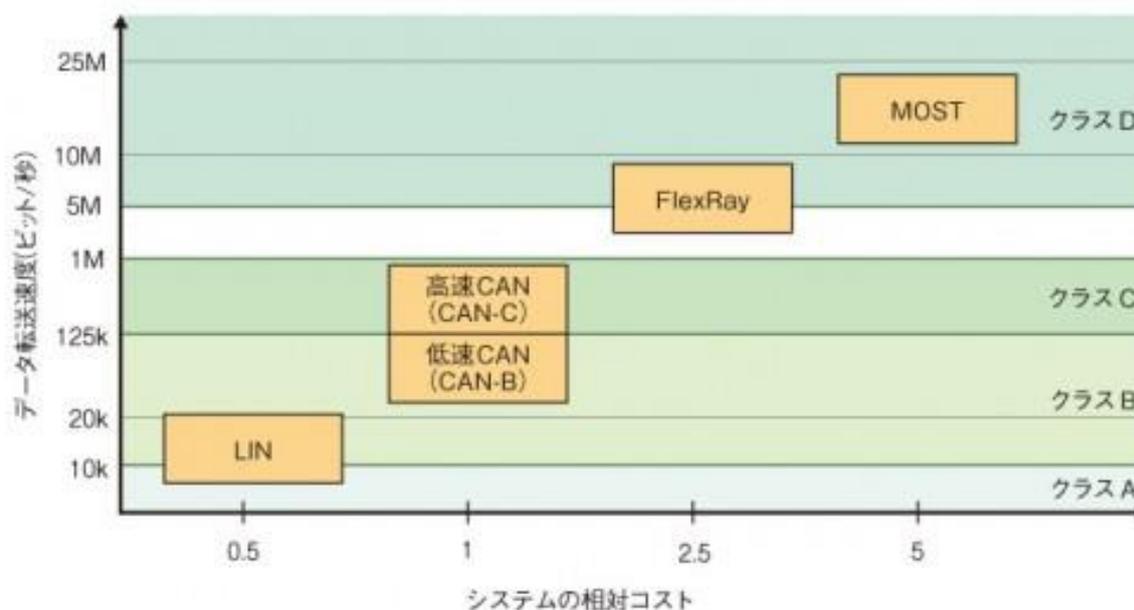



図 1.4-1 通信速度とシステムの相対コスト

1.4.1 CAN

CAN は、パワートレイン系、ボディ系、オーディオ/マルチメディア系などに幅広く適用されており、事実上の業界標準として、世界中の数多くの車種に採用されています。

現在一般的に使われている CAN の伝送速度は、最大 500k ビット/秒です。規格の上では、SAE[9]のクラス C に対応する最大 1M ビット/秒の高速 CAN（CAN-C）のほか、SAE のクラス B に対応する最大 125k ビット/秒の低速 CAN（CAN-B）があります。

そのほか、CAN には次のような特徴があります。

1. すべての ECU がマスターとして振る舞うマルチマスター方式を採用。
2. CSMA/CA（Carrier Sense Multiple Access/Collision Avoidance ）方式による、優先順位に応じたバス・アクセス。
3. ライン型のバス・トポロジを採用。
4. 差動電圧送信を利用して耐ノイズ性能を向上。
5. エラー検出機能とエラー時のハンドリング機能を装備。

1.4.2 LIN

LIN は、車両のバックボーン・ネットワーク以外の用途で、CAN ほど高いデータ伝送速度が求められない低コストのサブネットワークに向けて開発されました。CAN の 1/2～1/3 ほどのコストで済む、単線式ネットワークです。

[9] SAE は、モビリティ専門家を会員とする米国の非営利的団体です。SAE とは Society of Automotive Engineers の略。http://www.sae.org/

ドア・ユニット（ドア・ミラーやサイド・ウインドー、ドア・ロック）、エアコン、シート（座席）、サンルーフ、ヘッド・ライトなどの制御として、CAN のサブネットワークとして使用されるケースが多いです。

通信プロトコルとして、パソコンのシリアルポートなどにも使われている UART（Universal Asynchronous Receiver Transmitter）を採用し、電源には 12V の車載バッテリを使います。

そのほか、次のような特徴を備えています。

1. マスターが通信を制御するマスター/スレーブ方式を採用。
2. シングル･マスター方式を採り､バス上に 1 つだけ存在するマスター･ノードが通信スケジュールを管理。そのため通信の衝突を起こさない。
3. ライン型のバス・トポロジを採用。
4. 最大データ伝送速度は 20k ビット/秒。

1.4.3 MOST

MOST（Media Oriented Systems Transport）は、マルチメディア系に特化した車内ネットワークです。

MOST の規格では、物理層には光ファイバだけでなく、配線やメンテナンスが容易な銅線も利用できます。また､データ伝送速度は初期の規格では 25M ビット/秒（MOST25）でしたが､その後 50M ビット/秒（MOST50）､さらに現在では 150M ビット/秒（MOST 150）と高速化され、複数の動画の同時転送も可能としています。

そのほかの特徴は次の通りです。

1. 5.1 チャネル/7.1 チャネル・オーディオに対応。
2. バス・トポロジはリング型。
3. DVD ビデオなど、動画の伝送が可能。
4. プラグ・アンド・プレイに対応。
5. イーサネット・フレームを伝送可能で、インターネット通信に対応。

1.4.4 1394-Automotive

1394-Automotive は、民生機器向けの規格である「IEEE 1394」を基に、車載向けマルチメディア系ネットワークとして策定した規格です。高速伝送と低コストという特徴を備えています。量産車にはまだ採用されていませんが、今後車載ネットワークとして採用される可能性があります。

1.4.5 FlexRay

1990 年代半ば以降､車載ネットワークは CAN を中心に発達してきました。ですが近年では､ECU の増加に伴ってバスの負荷増大が危惧されはじめています。また、ステアリング（操舵）やブレーキなど、これまで導入が難しかったクルマの基幹機能にもネットワーク技術を応用しようという機運があります。こうした用途に対応できる車載ネットワーク規格として提唱されたのが「FlexRay」です。2000 年にコンソーシアムが設立され、ポスト CAN としても注目されています。

最大 10M ビット/秒での通信が可能で、物理層には光ファイバと銅線の両方が使えます。日本では CAN の後継プロトコルとして、2.5M ビット/秒や 5M ビット/秒の規格も策定されています。

そのほかの特徴は次の通りです。

1. タイム・トリガー通信を採用し、イベント送信にも対応。
2. バス・トポロジは、ライン型とスター型の混合。
3. 送信遅延時間を保証可能。
4. バスの冗長化や通信同期機能などによって、高い信頼性を確保。

リアルタイムOSの必要性

2.1 OSとは

OS とは、ソフトウェアが共通で行わなければならないハードウェアの制御など、基本的な部分を抽象化するソフトウェアのことをいいます。ここでいうハードウェアとはマイコンの周辺機器を指し、ハードウェアの制御とは入出力機能の管理、メモリやマイコン処理の管理などをいいます。

2.2 ＯＳ搭載時の問題点

① **OS を使用できる技術者の不足**

OS を使用する場合、以下に示すような新しい知識が必要となります。

- 共有している資源に対する排他制御（**6.4 排他制御** 参照）
- 処理ごとに割り当てる優先順位
- 提供される各種サービスコールの使用方法

さらに、組込み機器であるため**ハードウェア**を必須とし、**教育環境が不足**している状況も、技術者の不足という問題に拍車をかけています。

② **OS 対応開発ツールの不足**

OS が使用できる開発環境は、従来の開発方法に比べて充実していません。しかも、OS により処理が並列に動作するため、デバッグ[10]の際の**原因特定がより困難**になり、開発ツールへの要求は大きいといえます。

③ **ライセンス等のコスト**

OS を使用するためには、OS を開発しているメーカにライセンス料を支払う必要があり、ライセンス等のコストがかかります。ライセンスの形式はさまざまですが、開発プロジェクト単位で購入する方式、または使用する技術者数分だけ購入する方式が一般的です。OS によっては、OS を使用して開発した製品の製造した数や、販売した数に応じて料金を請求する「ロイヤリティ」という方式もあります。

④ **ソースコードを含めた情報公開義務**

「GPL（The GNU General Public License）」という GNU（GNU`s Not Unix）プロジェクトが規定しているライセンス方式が有名です。GPL は主に GNU プロジェクトが開発したソフトウェアに対するライセンス体系で、ソースコードの公開を原則としています。使用者がソースコードを含めた再配布や改変に対する自由を認めている代わりに、ソースコード内部に何らかの変更を加えた場合、そのソースコードを公開する必要があります [11]。

⑤ **使用する OS の信頼性**

OS のソースコードが閲覧できない場合、内部で OS が行っている処理を調べることは非常に困難です。この場合、もし OS のソース内にバグ[12]やトラブルの原因があれば、アプリケーションでいかに安全な設計をしようとしても、システムの安全性を保証することができません。

---

[10] コンピューターープログラムの不具合を発見し修正する作業のことです。
[11] GPL の詳しい情報は、GNU プロジェクトホームページ（http://www.gnu.org/）から「GNU 一般公衆利用許諾契約書」をお読み下さい。
[12] コンピューターープログラムの誤りや不具合のことです

このように、OS を使用する際にはいくつかのデメリットがあります。一方、以下のような大きなメリットもあります。

- OS がハードウェアを隠蔽することで、ハードウェアに依存しないアプリケーション開発を可能にする。
- タスク単位で分業化することで、開発効率の向上を可能にする。
- タスク単位でモジュール化することで、モジュールの再利用を可能にする。

これらを天秤にかけ、OS を利用するかどうか判断する必要があります。

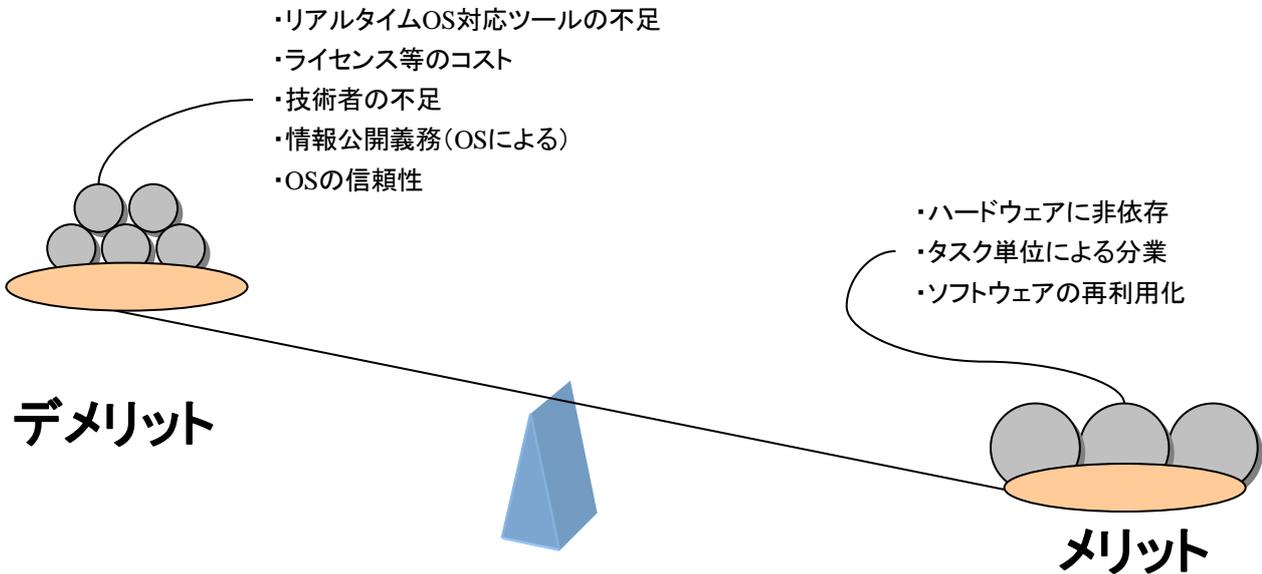



図 2.2-1 OS 利用の天秤

2.3 リアルタイム OS の概要

「リアルタイム」という言葉の意味は何でしょうか？「Real Time」という言葉を、英和辞書で調べると「同時、実時間」のように訳されています。「ニュースをリアルタイムにお届けする」という意味は、おそらくこのような意味合いで「提供している情報に遅れがない」ということを強調しているのでしょう。しかし、リアルタイム OS が指す「リアルタイム」とは、「同時」ではなく、「**一定時間内**に応答を返すことを**保証**している」ことを指しています。

マイコンが同時に実行できる処理は１つだけです。つまり、マイコンに対して複数の処理を与えても同時に処理されることはありません。どの瞬間で見てもマイコンは１つの処理だけを実行しているのです。しかし、リアルタイム OS を利用すると、見た目では**あたかも同時に**マイコンが複数の処理を実行しているよう見えるようになります（**図 2.3-1**）。これが、リアルタイム OS が、リアルタイムと呼ばれる理由です。

この「並列処理」には、次の項で説明するリアルタイム OS の説明の「B さんが複数の処理を与えられたとき、その処理の重要度で、行わなければいけない作業を判断する」という方法が取られます。つまり、今マイコンが処理しなければならないものを判断して、マイコンに処理を渡すのです。この方法を使用すれば、ある一瞬だけを見ると「1 つの処理を 1 つのマイコンが実行している」のですが、長い時間で見ると「複数の処理を 1 つのマイコンが処理をしている」ように見えるのです(**図 2.3-2**)。

リアルタイム OS は上記の理由などから、「同時」には実行できないのですが、「重要な処理が一杯入っていたので、遅れて実行してしまった」というのでは困ります。OS 利用者からすれば、「同時」ではなくても操作上「同時に」動いて欲しいので、応答する時間が重要です。

リアルタイム OS 非使用

CPU は必ず一つの処理を実行後に、次の処理に移るため、複数の処理を同時に実行させることは出来ません

リアルタイム OS 使用

各処理をリアルタイムに切り替えることによって、CPU に複数の処理を実行させることができます。

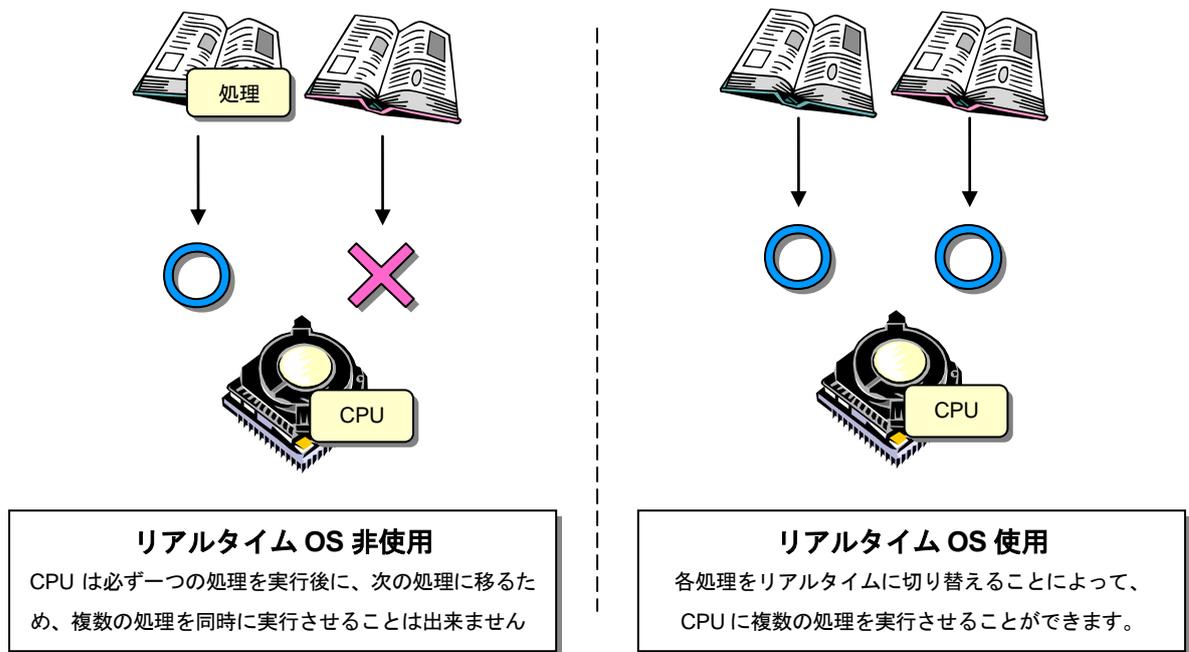



図 2.3-1 処理方法の違い

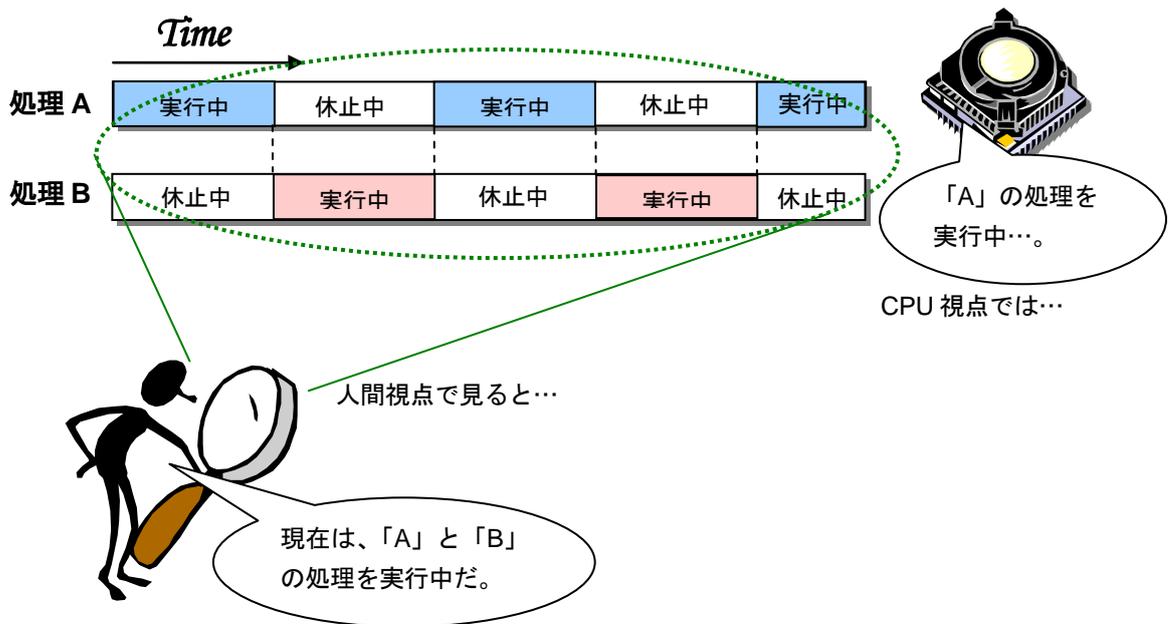



図 2.3-2 マイコンと人間が見た「実行中」の違い

ここで、「あれ、**スピードが一番重要**ではないの？」と考える人がいるかも知れません。確かに、「同時」という風に考えると、できるだけ高速な処理を行って、次の処理に移ればよいと考えがちです。しかし、リアルタイム OS にもっとも重要なのは、「時間」なのです。たとえば、「平均的には非常に高速で複数の処理を的確に行うが、ある条件下では非常に遅く、応答時間は不定という OS」と、「平均的にはやや遅いが、処理・応答される時間は保証されている OS」とのどちらがリアルタイム OS と呼ぶのに適当でしょうか？

リアルタイム OS は、主に制御機器などの用途に用いられます。そこでは、周期的に一定の間隔で行う処理や、緊急時に優先的に行わなければならない処理が多くあります。このとき、重要になるのが「タイミング」つまりは、「時間」なのです。たとえば、衝突事故などで人命を守るエアバッグが、「ある時は高速に開くが、ある条件下では不定のタイミングで開く」というのでは、安全とはいえないでしょう。しかし、「遅くとも確実に人命を守れる時間以内に開く」というのであれば、時間面では「安全」といえます。

このように、用途の面から見ても、リアルタイム OS に要求されるのは「必要な処理時間、応答時間を守ってくれること」なのです [13]。

つまり、リアルタイムとは「**一定時間** [14] **内に応答を返す**」ことを言い、一定時間内に応答を返すことができる OS をリアルタイム OS と呼ぶのです。

いわゆる OS というと、パソコン上で動作する Windows、Mac OS を思い浮かべる方が多いでしょう。これらの OS は複数の処理タスクを管理する機能を有してはいますが、一般的には複数のタスクを一定時間ごとに自動的に切り替えるようになっています。緊急性の高い処理を行おうとしても、他のタスクが処理途中である場合には即時実行されない場合があります。

リアルタイム OS の場合は、何かの処理途中に緊急性の高い処理を行う必要ができた時、実行していた処理を中断して緊急性の高い処理を即時実行することができます。このように、リアルタイム OS には一定時間内に応答を返すための機能を有しています。また、どの処理が重要なのかを示すための重要度のことを「優先度」と呼び、各タスクがそれぞれの優先度を持つことができます。この優先度を基にタスクの切替えを行います。

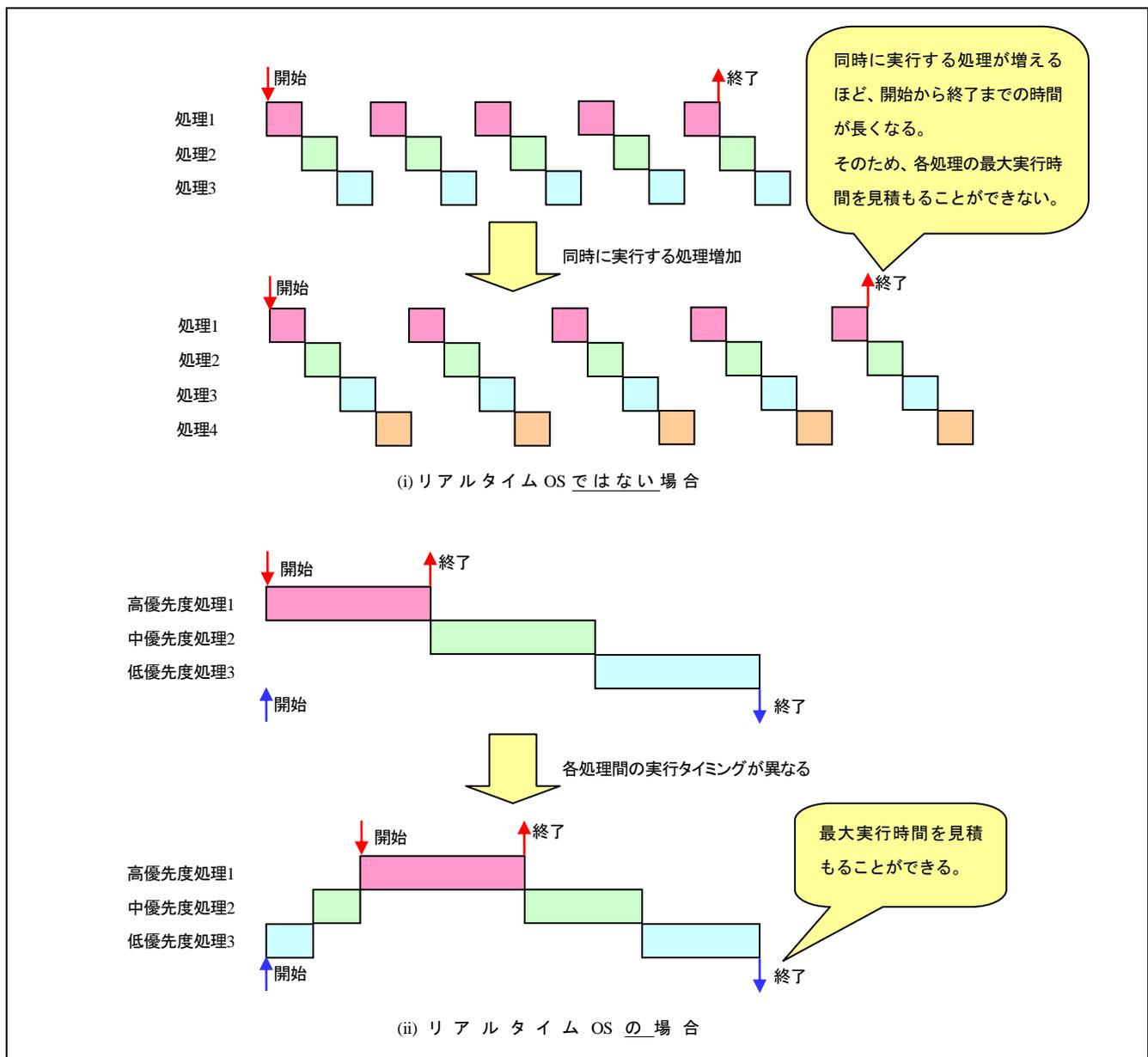



図 2.3-3 汎用 OS とリアルタイム OS の動きの違い

13 確実でない場合でもリアルタイム OS と呼びますが、その場合は最大処理時間が予測できる必要があります。
14 ここでいう一定時間とは、処理が完了するまでの最大時間（その時間以内に確実に処理を終えれる時間）のことです

2.4 並行処理単位とは

リアルタイムで取り扱われる処理は、並行して処理が動作することになります。この「並行処理単位」を「**タスク**」といいます。つまり、「LED を点滅させる処理を『LED タスク』という」と決めた場合、「LED タスク」とは「LED を点滅させることを並行で行うことができる処理」という意味になります。

この「タスク」という言葉は、正確にはメモリ領域の取扱い等によって「プロセス」と「スレッド[15]」という言葉で区分できます（**図 2.4-1**）。

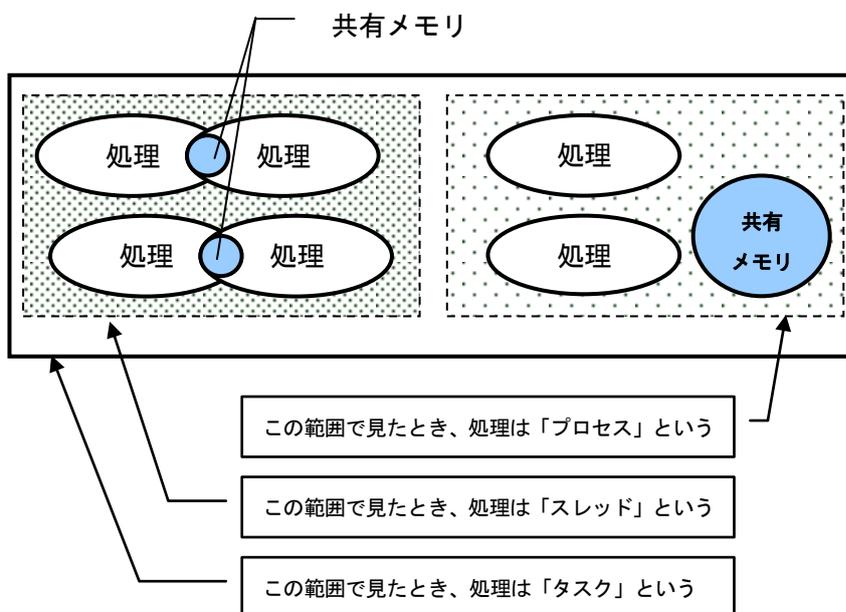



図 2.4-1 メモリの取扱いによる区分

表　2.4-1 モデルとその特徴

モデル	メモリ			応答速度
	取扱い	使用サイズ	保護	
プロセスモデル	独占	大きい	有り	遅い
スレッドモデル	共有	**小さい**	無し	**速い**

●プロセスモデル

メモリ保護機構がある。別プロセスのメモリが見えないためプロセス間の独立性が高い。メモリのサイズが大きくなり、応答性に限界がある。プロセス中にさらに複数のスレッドを持てるものが一般的。例として、Windows、Linux、OS-9、QNX がある。

●スレッドモデル

メモリ保護機構がない。別スレッドのメモリにアクセスできるため、スレッド間の結合性が高い。メモリサイズが小さく、応答性が高い。例として、**OSEK**、VxWORKS、NUCLEUS がある。

[15] 本来、OSEK は、スレッドモデルであるため並行処理単位の名称は「スレッド」ですが、仕様書中で「タスク」と名称を統一しているため、本書でもそれに従って「タスク」と呼びます。

2.5 リアルタイム OS を使用するメリット

2.5.1 ハードウェアの抽象化

ハードウェアの管理を OS が行ってくれることにより、アプリケーションではそれぞれがハードウェアを意識した制御プログラムを持つ必要が無く、もしハードウェアを制御する必要がある場合には、OS が提供するサービスコールを使用して、OS にハードウェア制御を代行してもらいます(**図 2.5-1**)。

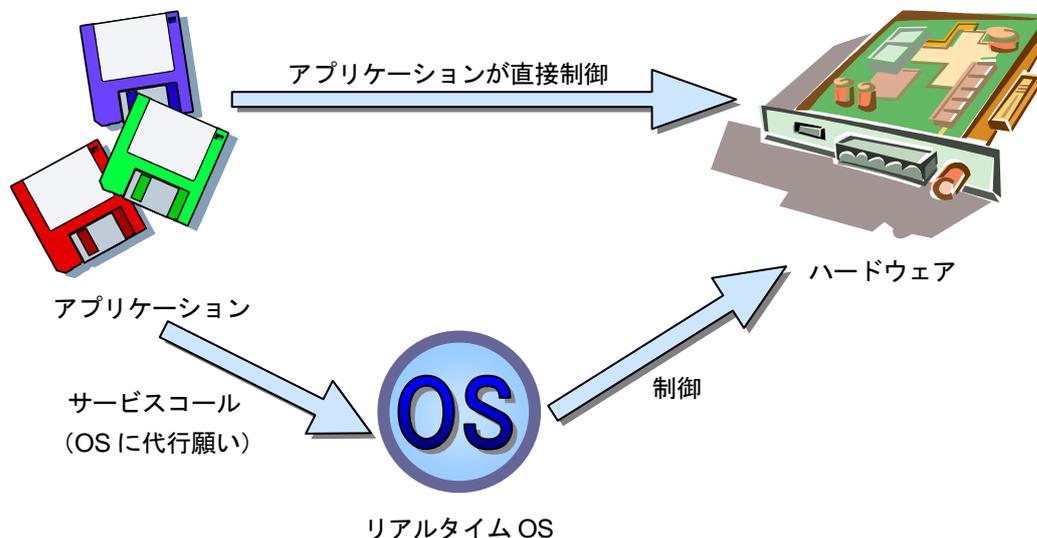



図 2.5-1 ハードウェアを制御する２つの方法

たとえば、私たちが荷物を届ける際に、自分自身で直接荷物を届けると、非常に手間と時間がかかりますが、宅配センターに連絡して、「荷物を届けて欲しい」と連絡をすれば、荷物を代わりに届けてくれます。これと同じように、アプリケーションが直接ハードウェアを制御する手間を省くために、OS が制御を代行してくれるのです。

アプリケーションがハードウェアを制御する必要がないと、ハードウェア上で作成するアプリケーションの負荷が減ります。また、OS が異なるハードウェアでも対応していれば、異なるハードウェアでもアプリケーションが動作するようになります。このようにアプリケーションがハードウェアを意識せずに作成できるようにすることを「ハードウェアの隠蔽化」といいます。

2.5.2 プログラム開発の分業化

プログラムは、実際には複数の小さな処理にわかれています。

携帯電話の例でいえば「電波状況の調査」、「時計時刻の更新」、「通話時間の管理」などです。これら複数の機能は、あるときだけ行わなければならないものや、周期的に行う必要があるものなど、実行条件はさまざまであるため、プログラムは各処理が他の処理を監視、制御する形で最適な順番に実行していきます。OS を用いていない場合、1 つの大きな処理単位を繰返し実行するしかないため、アプリケーションは小さな処理として完全に分離することが出来ません。また、新たに処理を加えた場合にも**全体的に処理順序や処理頻度を調整し直す**必要があります。

OS を導入すると、監視、制御の部分を代行して貰うことが可能になります。すると、複雑に絡み合っていた小さな処理は､自身が行わなければならない処理だけに集中することができ､独立した処理として実行できます。また、独立した処理であれば、各処理の追加・削除を行った場合でも、**システム全体への影響は少なく**済みます(**図 2.5-2**)。

OS を用いない開発では、複数人が行っていた作業に対して、全体への影響を調整しながら作成する必要がありました。しかし、OS を導入することでプログラム中の処理を独立した処理として作成することができるようになると、同時に複数人で開発を進めることができます。このように、処理ごとに作業を分担することを「**分業**

化」と言います(図 2.5-3)。

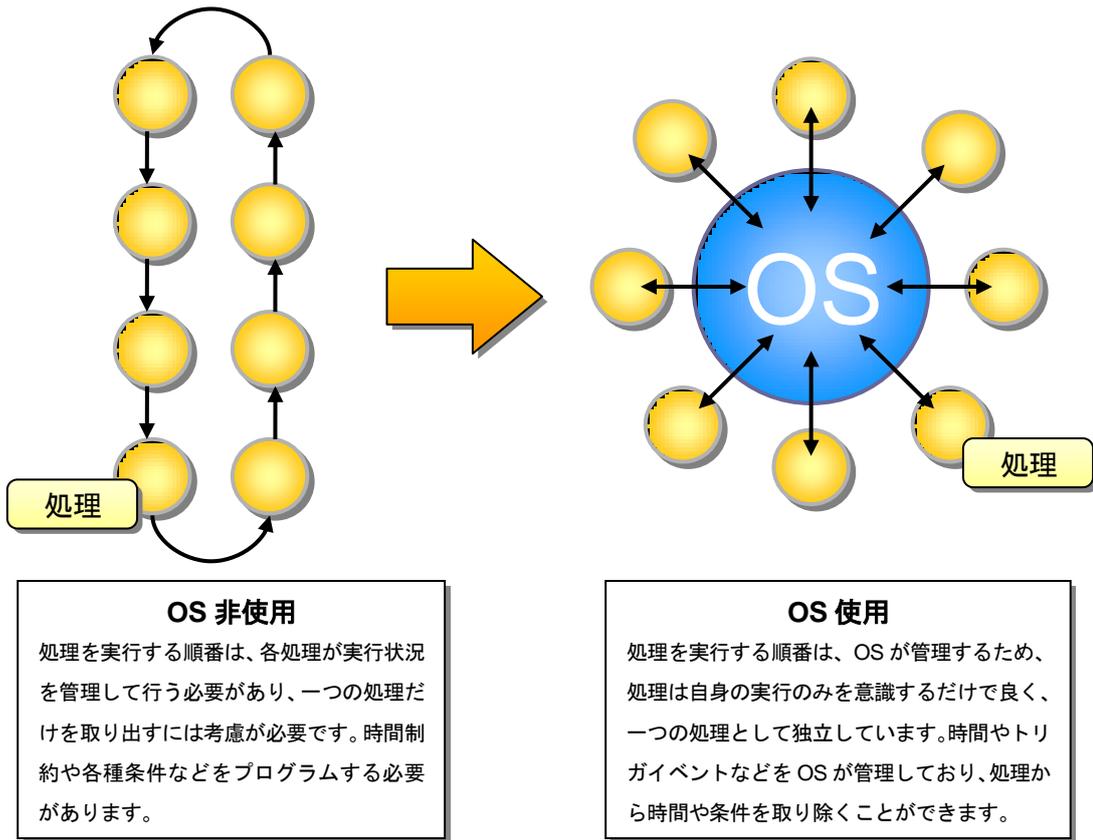



図 2.5-2 OS 使用による処理の取扱い

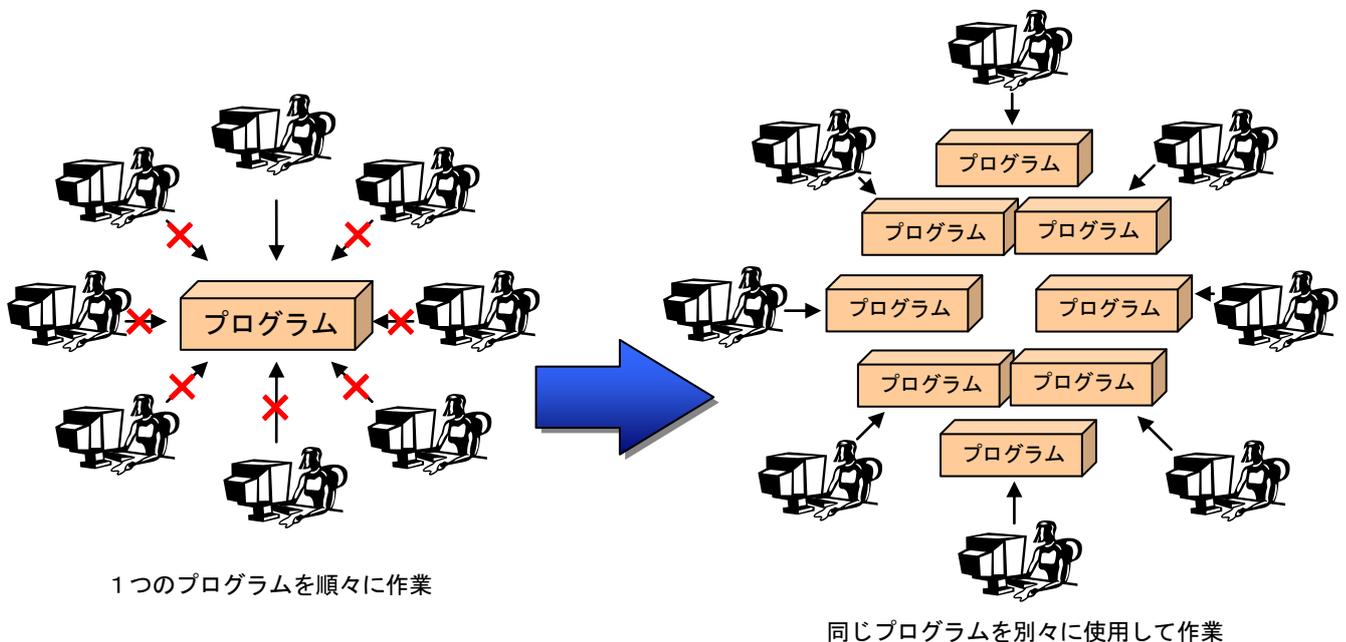



図 2.5-3 プログラム開発の分業化

2.5.3 実行時間管理

前述した「ハードウェアの抽象化」、「プログラム開発の分業化」は OS 全般に関するメリットです。さらに「リアルタイム OS」としてのメリットとして、「実行時間管理」があります。

リアルタイム OS に求められるのは応答時間制限の保証です。複数のタスクを同時に処理する際、それぞれのタスクについての完了時間制約を守るように各タスクの実行時間の管理をします。

単純に全タスクを一定時間毎に処理を切り替えた場合、本来は早く完了してほしいタスクも、他のタスク処理が間に入ってしまっているために完了が遅れてしまうかもしれません。そこでリアルタイム OS は、早めに完了させたいタスクの処理優先度を上げる、処理時間の割合を他のタスクよりも大きくする等を行い、制約時間内にタスク処理が完了するように実行時間を管理します。

TTV(タスク・トレース・ビュー)の実習

リアルタイムOS を搭載したシステムにおけるソフトウェアの動きをシミュレートするもので、マウス操作でタスクの状態を変更できるようになっています。

http://www.t-engine.org/ja/wp-content/themes/wp.vicuna/html/ttv/files/ttv.html

★TTV シミュレーション

2.6 リアルタイム OS を使用するデメリット

2.6.1 速度的なオーバヘッド

リアルタイム OS を使用した場合、OS を使用しないプログラムに比べ、OS が動作する分だけ**実行効率は低下**します。これは、プロセッサの利用率が高い場合には、処理時間の遅れが発生する原因になります。

たとえば、あるプログラムを OS 無しで実行した場合と、OS でタスクとして動作させた場合を考えてみます。対象となるプログラムは 3 つの処理にわかれているプログラムで、リアルタイム OS を使用したプログラムでは、各処理をタスクとして作成するとします(**図 2.6-1**)。

この場合、リアルタイム OS を使用しない場合は、直接次の処理を実行するため、次の処理へ移る時間に遅延はありません。しかし、**リアルタイム OS を使用する場合**、OS が処理の終了や、次の処理への受け渡しを行うため、**遅延が発生**します。

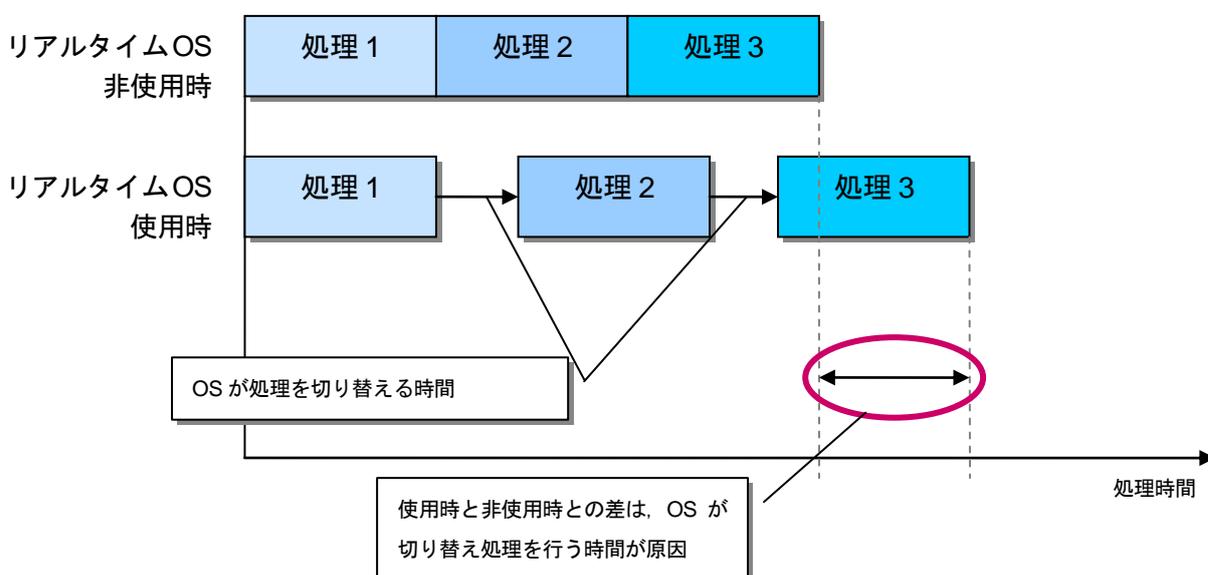



図 2.6-1 OS 使用による処理の遅延

マイコンによっても異なりますが、この遅延は数μ秒から 100μ秒といわれています。近年では、マイコンの高速化が進んでいるため、タスク切り替えによる遅延はそれほど問題視されていませんが、小規模なシステムでハードリアルタイムを求めている場合には、OS を使用せずタイマ割込みなどを使用したシステムのほうが効率的である場合があります。

また、上記の場合以外にも、OS 内部に存在する「割込み禁止区域（連続して実行する必要がある処理）」という問題があります。リアルタイム OS であっても、リアルタイム性を実現する処理（ディスパッチ等）を行っている最中は、割込みの受付を禁止します。システム実現のために最低限必要とする処理であるため、構造上仕方ありませんが、これが原因で処理自体が遅延してしまうことになります。

2.6.2 スタック領域の増加

OS の利用において、リソース面で問題になるのが「スタック領域の増加」です。
リアルタイム OS では、**タスクごとにスタック領域を渡す**必要があります。当然、処理自体のスタックは OS 使用の有無に関係なく発生しますが、タスクごとに渡されるスタック領域を OS がタスク切り替えの際等に使用するため、実際の容量よりも多くの容量が必要となります。

タスクごとにスタック領域を持つ必要がある理由は、OS を使用する場合、タスクの優先度や実行順が途中で変化する可能性があり、単純に前の処理を記憶しておくだけでは情報が足りないからです。どのように順番が変

わっても、あるタスクに処理が戻る場合には「**そのタスクがどのような状態だったか**」を記憶しておく必要があります。詳細は**3.4.1 スケジューリング方法の違い** を参照して下さい。

2.6.3 排他制御が必要

OSEK 等の**スレッドモデル**で実現されるリアルタイム OS の場合、ハードウェア上に存在するメモリなどのリソースは、**共有して使用**されます。しかし共有とはいえ同時に同一メモリにアクセスなどが行える訳ではありません。複数のタスクが処理を行う場合、このリソースの同時使用は発生する可能性が常に存在するため、プログラム上でこのリソース同時使用を防がねばなりません。

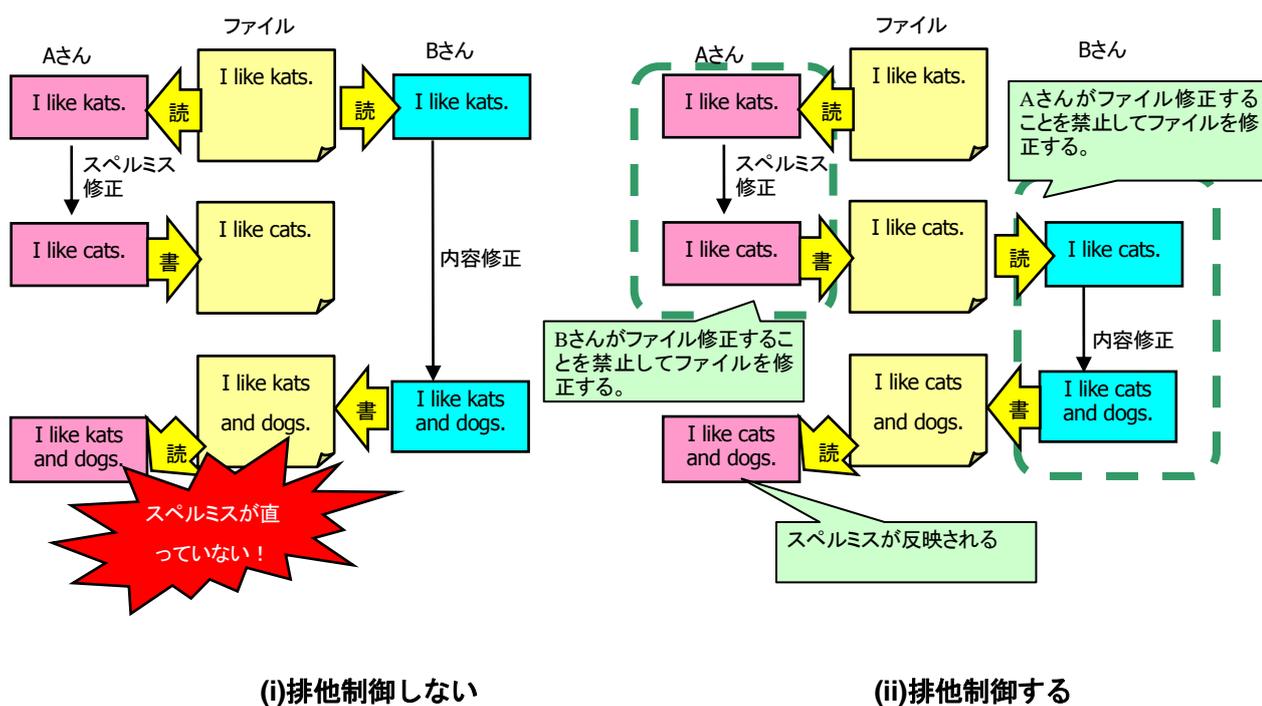
それらを実現するために、リソース使用時には他の処理にリソース使用を禁止する制御を行う必要があります。それを「排他制御」といいます。
　排他制御を行うと、高優先度タスクの処理が低優先度タスクに待たされ処理時間が長くなることがあります。また、プログラムが複雑になる傾向があり、排他制御に関連するバグは後を絶ちません。

例として A さんと B さんが同時にファイルを修正することを考えます（**図 2.6-2**）。排他制御をしない場合、A さんと B さんが同時にファイルを修正すると A さんの修正が反映されないということが起こり得ます。しかし、排他制御をする場合、どちらかが修正中はもう片方は修正することを禁止するので A さんと B さんの修正内容がきちんと反映されます。ただし、片方が修正している間、もう片方は修正することを待つことになるため、その分排他制御しない場合に比べ処理速度は長くなります。

(i)排他制御しない　　**(ii)排他制御する**

図 2.6-2 排他制御の例

2.7 割込み処理

2.7.1 割込みとは

マイコンには「割込み」という処理が存在します。

ここでいう割込みとは、外部（周辺機器）からの何らかの要因により、CPU の処理を一時中断して別の処理を実行することです。割込みの要因の例として、スイッチの押下、設定したタイマの満了等があります。割込み処理が終わると、中断した処理を再開します。

日常生活で例えると、電話の着信が割込みの外部要因、電話での通話が割込み処理にあたります。

図 2.7-1 日常生活における割込み処理は本処理（業務）中に電話がかかってきた時の処理の流れを示したものです。

マイコンにおける割込みもこの図と同じように、割込みの要因が発生したら現在の処理を中断し、割込み処理を終えた後、中断した本処理を割り込まれた所から再開します。

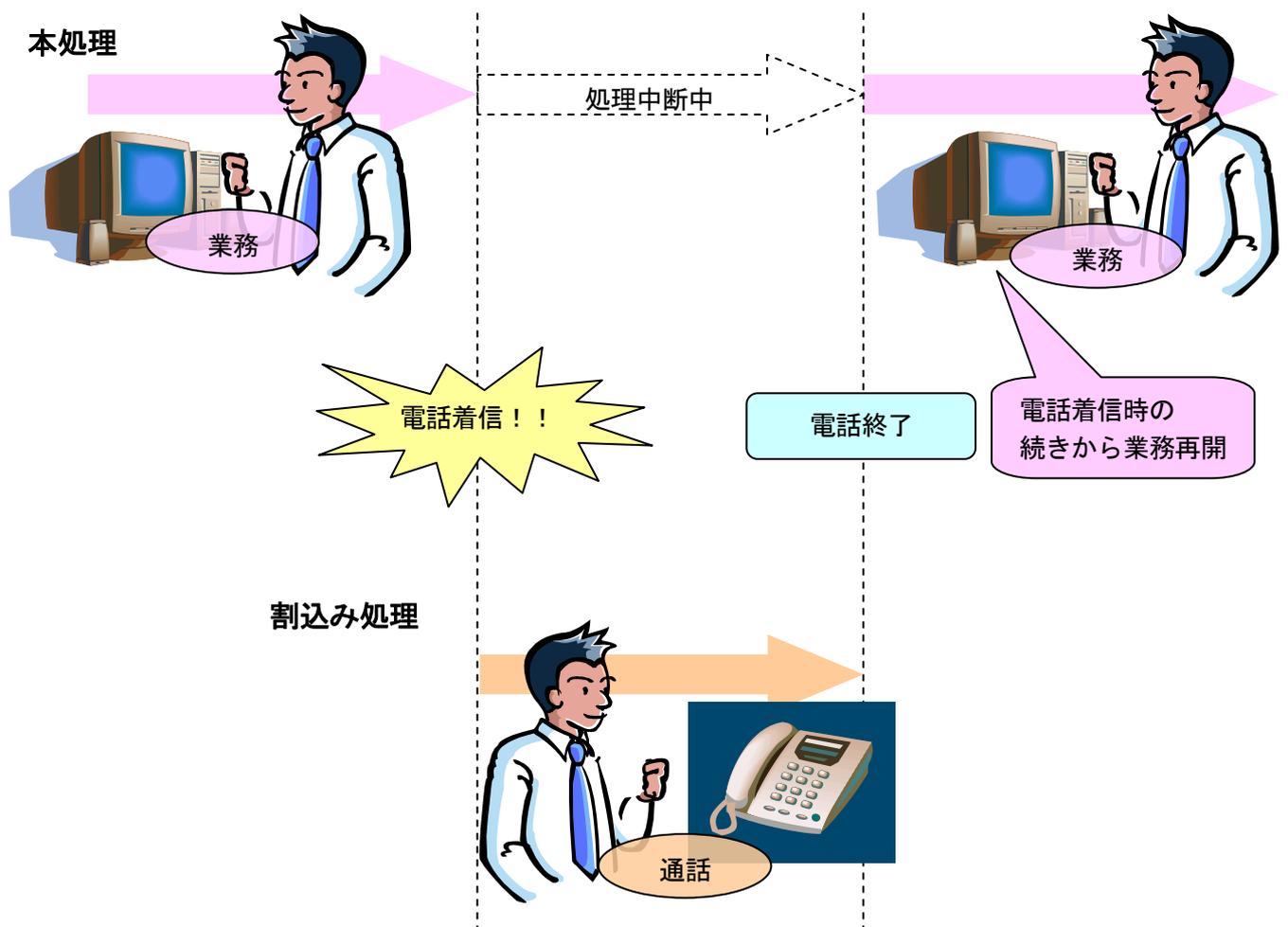



図 2.7-1 日常生活における割込み処理

ここで重要なポイントとして、割込み処理終了後に「割り込まれた時点から処理を再開」するという点があります。C 言語等でいう関数呼出しとは違い、どこで割り込まれるかは決まっていません。

図 2.7-1 日常生活における割込み処理の例のように、通話終了後に業務を続きから再開するにはどうしたら良いでしょうか？

続きはここからという印を付ける、頭で覚える等、方法はいろいろありますが、続きから再開するためにはその時の状況を「記録」もしくは「記憶」する必要があります。

マイコンにおいてもこの点は同じです。割込み要因が発生した際に「どこで割り込まれたか」「その時点でどん

な値を格納していたか」というその時の状況について何らかの方法で記憶する必要があります。割込み処理が終了した時に、記憶されていた情報を基に元の処理へ戻ってくるのです。

2.7.2 割込みの仕組み

割込みの大まかな流れについて日常生活を例に説明しましたが、マイコンとして実際にどのように割込み処理を行うのか、もう少し内部を見てみましょう。

割込みは、周辺機器からの処理要求を CPU に伝えることによって起こります。この要求信号を管理するために、マイコンには「割込みコントローラ」と呼ばれる制御回路が存在します。
割込みコントローラは周辺機器と CPU の間に接続されており、周辺機器からの要求に対して CPU にそれを伝えるかどうかをここで判断します。必ずしも全ての要求を CPU に伝えるわけではなく、各周辺機器に対する割込み許可/不許可を設定でき、その設定に応じて必要な割込み信号だけを CPU に伝えます。

割込みコントローラは、CPU へ割込み要求を伝える際に「どの機器からの割込みか」という割込み要因情報も伝えます。割込み要因情報は番号で伝えられ、これを参照することにより、どの割込み処理を行うかの判断を行います。
また、マイコンはどの割込みでどの処理を行うかを決める「ベクタテーブル」と呼ばれるものを持っています。
ベクタテーブルは、各割込み要因に対応する割込み処理の場所情報を一覧として持つものです。割込み発生時には割込み要因番号とベクタテーブルを照らし合わせて、実際に行う割込み処理を決定します。

割込みコントローラとベクタテーブルの働きについて**図 2.7-2** 割込み処理の流れに示します。

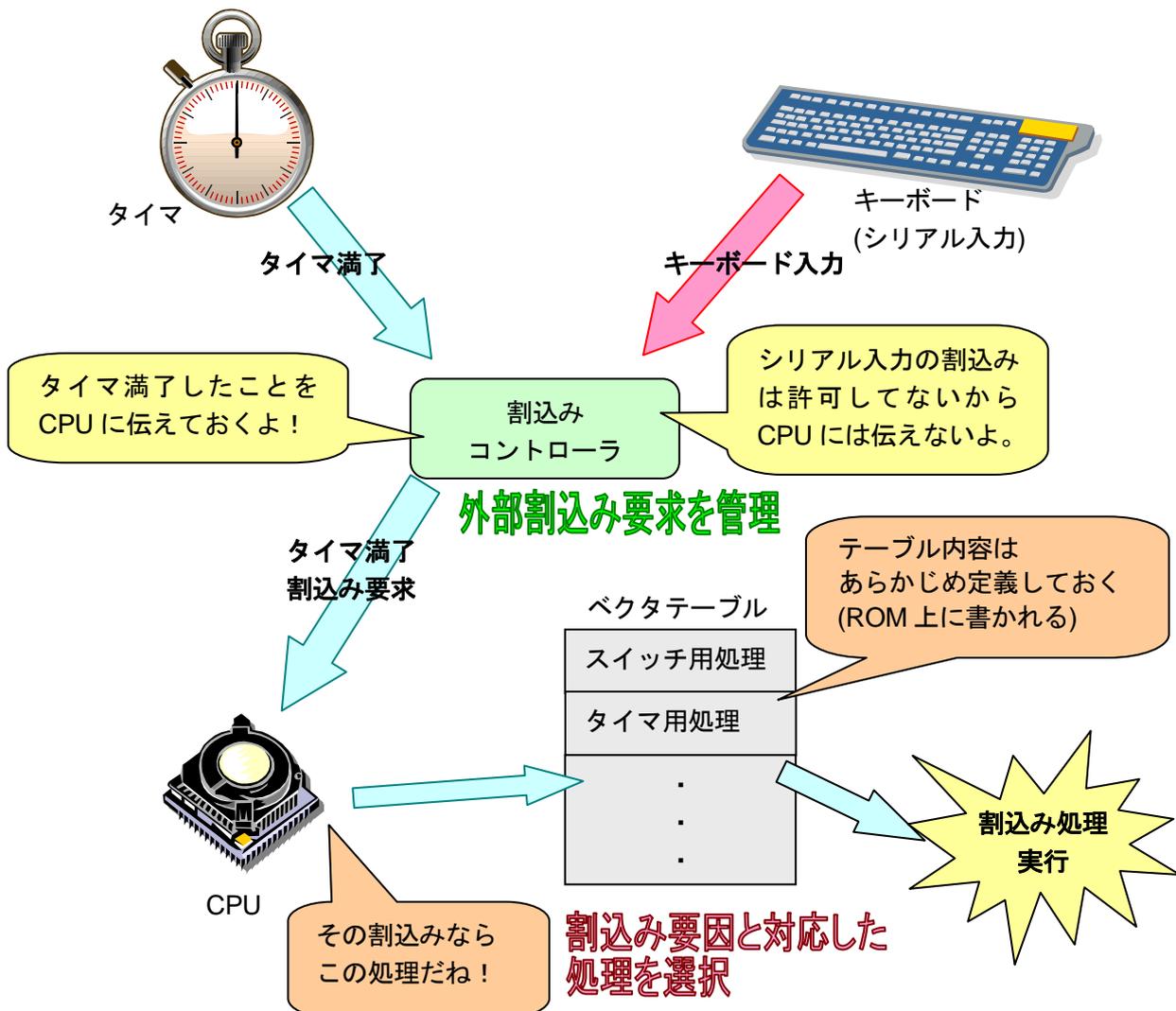



図 2.7-2 割込み処理の流れ

2.7.3 OS における割込み処理

リアルタイム OS でも割込み処理は深く関わってきます。

割込み処理が終わった後、本処理の続きから再開するということを述べましたが、OS ありの環境の場合はそうなるとは限りません。

2.4 並行処理単位とは で OS における「タスク」という単位について説明しましたが、タスクの状態や優先度が割込み処理中に変化する可能性があります。割込み処理中にそのような変化が起こった場合にどうなるのか考えてみましょう。

図 2.7-3 OS ありの環境での割込み処理一例のような簡単な例を基に、OS ありの環境での割込み処理が及ぼす影響を見てみましょう。

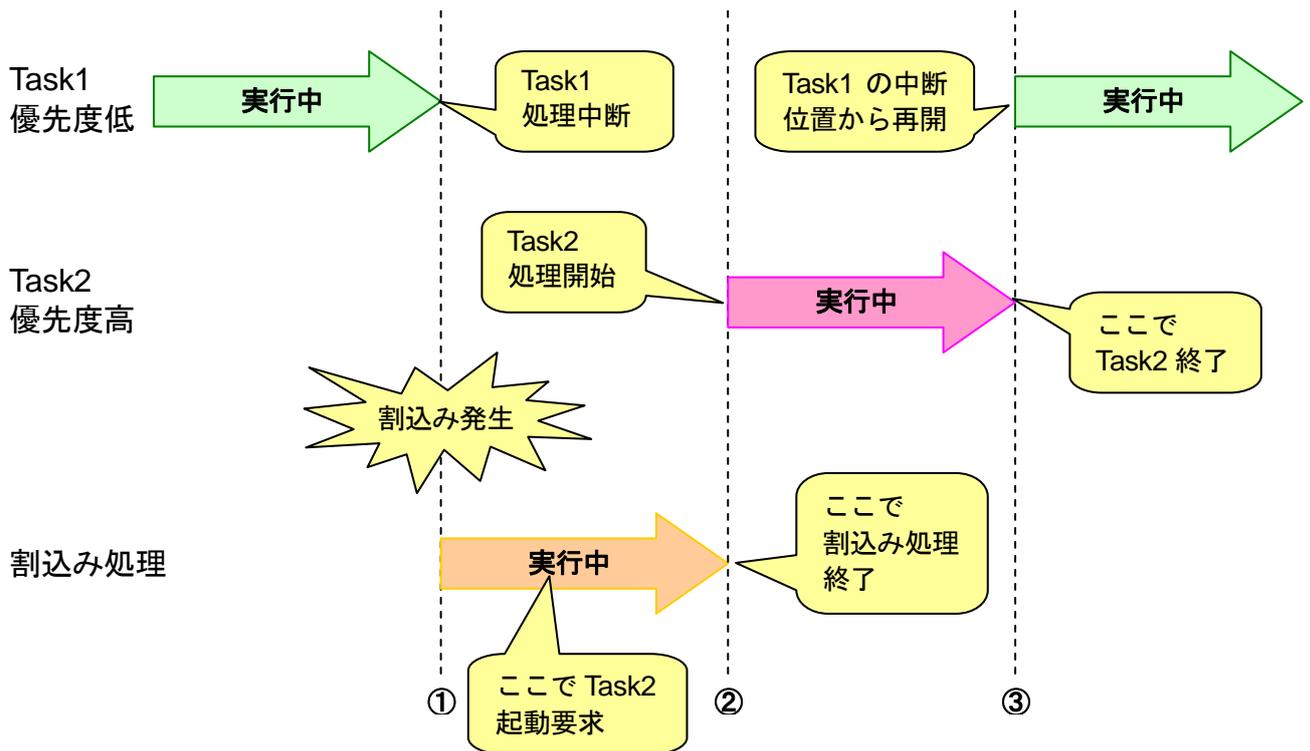



図 2.7-3 OS ありの環境での割込み処理一例

図 2.7-3の場合、Task1 実行中に図の①の時点で割込みが発生するため、Task1 の処理が中断されます。OS なしであれば、割込み処理完了後、本処理である Task1 の中断箇所から再開します。

しかし、今回の場合は割込み処理中に Task2 の起動要求が行われています。さらに、Task2 は Task1 よりも優先度が高いので、割込み処理が終わる②の時点で、Task2 の処理が実行されます。

Task2 の処理が完了する③で、①の時点で中断されていた Task1 の処理を再開します。

OS なしと違い、割込み処理終了後に元いた処理（Task1）とは違う処理（Task2）が実行されています。さらにその別処理が完了後に、割込みで中断された処理が再開されます。

OS なしの場合は単純に割込み発生箇所の状況を記憶しておくだけで、割込み処理後の復帰が出来ましたが、OS ありの場合はそれだけでは管理しきれません。

図 2.7-3の例の場合に必要な処理を考えてみましょう。

以下のような処理をしなくてはなりません。

- ①の割込み発生時点で実行中だった Task1 の状況（実行位置、変数）の記憶
- ②の割込み処理終了時点でどのタスクを実行するべきかの判断
- ③の Task2 処理終了時点でどのタスクを実行するべきかの判断

Task2 の実行中でも、Task1 の状況をずっと記憶しておく必要があります。今回の例では Task1 だけですが、状況によってはもっと多くのタスクの情報を記憶しなければならない場合もありますし、優先度が途中で変わると、実行順が入れ替わる可能性もあります。また、処理中断中だったタスクが他のタスクによって強制終了されることもあります。

途中でタスクの状態や優先度が変化するため、タスクや割込み処理が終了する度に、残りの起動中タスクのどれが一番高優先度で処理をしなければならないかという判断が必要になります。従って、OS なしのように単純に前の情報だけ記憶するだけでなく、各タスクの情報を OS は管理しなくてはなりません。

3 OSEK/VDX 仕様概論

3.1 OSEK/VDX とは

OSEK（Offene Systeme und deren schnittstellen fur die Elektronik im K raftfahrzeug）は、ドイツの自動車・電装メーカを中心に、ECU 関連の規格の標準化を行うプロジェクトとして 1993 年に発足しました。
VDX（Vehicle Distributed eX ecutive）はフランスの自動車メーカによって検討されていた OSEK と同様のプロジェクトです。
1994 年に、両プロジェクトが協調路線をとり、OSEK/VDX として 1995 年 10 月に仕様を提示しました。
主な開発成果は、車載機器制御用 OS の国際標準 ISO 17356 シリーズとして ISO が発行しています。

3.2 OSEK/VDX の特徴

OSEK/VDX 仕様は大きく 3 つのパートにわかれおり、OS、COM（Communication）、NM（Network Management）から構成されます(**表 3.2-1**)。

表 3.2-1 OSEK/VDX 仕様 1

OSEK OS	自動車制御用に機能を特化し絞り込んだリアルタイムカーネル仕様 □ 基幹ソフトであり、以下のような機能がある ・各種のシステムサービス提供 ・OSEX/VDXモジュールの管理
OSEK COM	CANを想定した自動車制御ネットワーク用通信プロトコルとそのAPI仕様を規定 ECU内通信のAPI仕様を規定
OSEK NM	CANを想定した自動車制御ネットワーク用のネットワーク管理手法とそのAPI仕様を規定

またその他に、OIL（OSEK Implementation Language）、ORTI（OSEK Runtime Interface）、OSEK timeOS、FT-COM（OSEK time Fault-Tolerant Communication Layer）、MODISTARC などがあります(**表 3.2-2**)。

表 3.2-2 OSEK/VDX 仕様 2

OIL	OS、COM、NMのコンフィギュレーション記述言語の仕様を規
ORTI	デバッガやICEが、OSEK仕様準拠のOSの内部情報にアクセスするためのインタフェース記述方法を規定
OSEK timeOS	タイムトリガ型のネットワークに接続するECUのためのタイムトリガ型のOS仕様について規定
FT-COM	タイムトリガ型のフォールトトレラントネットワーク※上で用いるための通信プロトコル仕様について規定
MODISTARC	OSEK/VDX仕様の検証手法について規定

※一部に故障が生じても処理が停止しないように設計されているネットワーク。
処理装置や補助記憶装置を二重化したりなどの対策がある。

3.3 OSEK OS の特徴

OSEK OS は自動車制御用に特化し機能を絞り込んだリアルタイムカーネル（以後 RTOS と記載）です。

他の RTOS より特に「厳格なリアルタイム」「低資源（メモリーリソース）」「スケーラビリティ[16]（大規模システムから小規模システムまで）」「信頼度」「コスト（低価格、オープンソース、ライセンスフリー）」を重要視しており、以下のような特徴があります。

- コンフォーマンスクラス（**3.4.2 コンフォーマンスクラス** 参照）と複数のスケジューリング方式によるスケーラビリティの確保
  ・8bit プロセッサでも動作
- 静的設定により可能となる最適化を活用
  ・カーネルオブジェクトは静的生成
  ・OIL によるコンフィギュレーション記述
- 2 つのレベルのエラー処理をサポート
  ・開発・テストフェーズ用と製品フェーズ用

3.4 OSEK OS の仕組み

3.4.1 スケジューリング方法の違い

マイコンに対してもっとも効率よく複数の処理を実行させるためには、各処理の実行を調整する必要があります。これを「**スケジューリング**」といい、スケジューリングを実現する OS の部位を「スケジューラ」といいます。OS が行うスケジューリングの代表的な例に「タイムシェアリング（TS）」と、「イベント駆動型」の二種類があります（**図 3.4-1**）。

タイムシェアリングは、処理ごとに実行時間を調整するもので、時分割と呼ぶこともできます。OS は、処理を実行するマイコンの時間を設定し、その時間を超えると次の処理へ実行を移します。優先的に実行させる必要がある処理は、実行時間を長く設定するようにして、優先度を変更します。

対して、イベント駆動型はイベントが発生した時点で、別の処理へ実行を移す方式です。処理には優先度を設定し、現在実行している処理より高い優先度の処理が発生した場合は、実行を優先度の高い処理に移します。低い優先度の処理は、実行する必要があっても優先度の高い処理が終了、または停止するまで実行することはできません。優先度の高い処理が、実行中のタスクに割り込んで実行されることを「**プリエンプティブ**」といいます。

図 3.4-2にプリエンプティブとノンプリエンプティブの作業の違いを示します。

OSEK OS は、**イベント駆動型**の方法を用いて、各処理をスケジューリングしています（**図 3.4-2**）。

---
[16] システムの拡張性。

図 3.4-1 スケジューリング方法の違い

ノンプリエンプティブ

仕事は後で

突然の仕事(高優先度)の依頼発生

飲み物を飲む

まだ飲み終わってないので飲む

飲み終わったので仕事に移る

プリエンプティブ

こっちが先だ

飲み物を飲む

飲み物を中断して仕事を優先する

仕事が終わったので残りを飲む

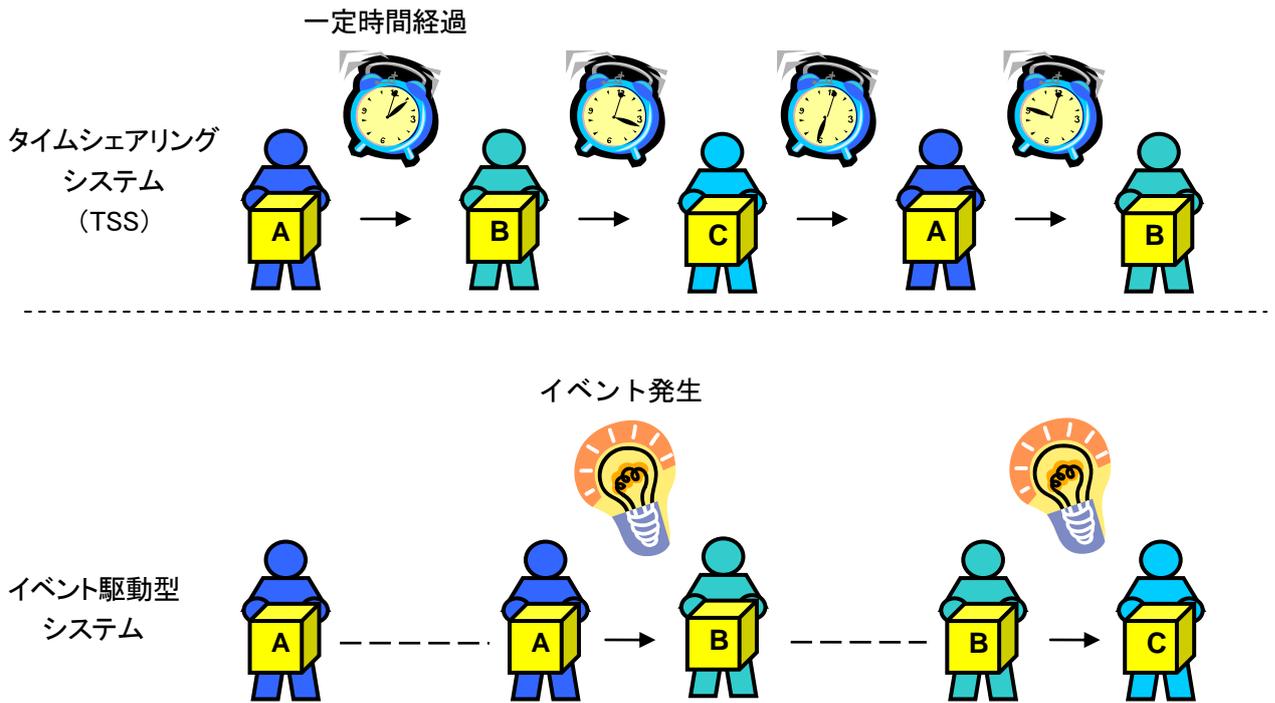
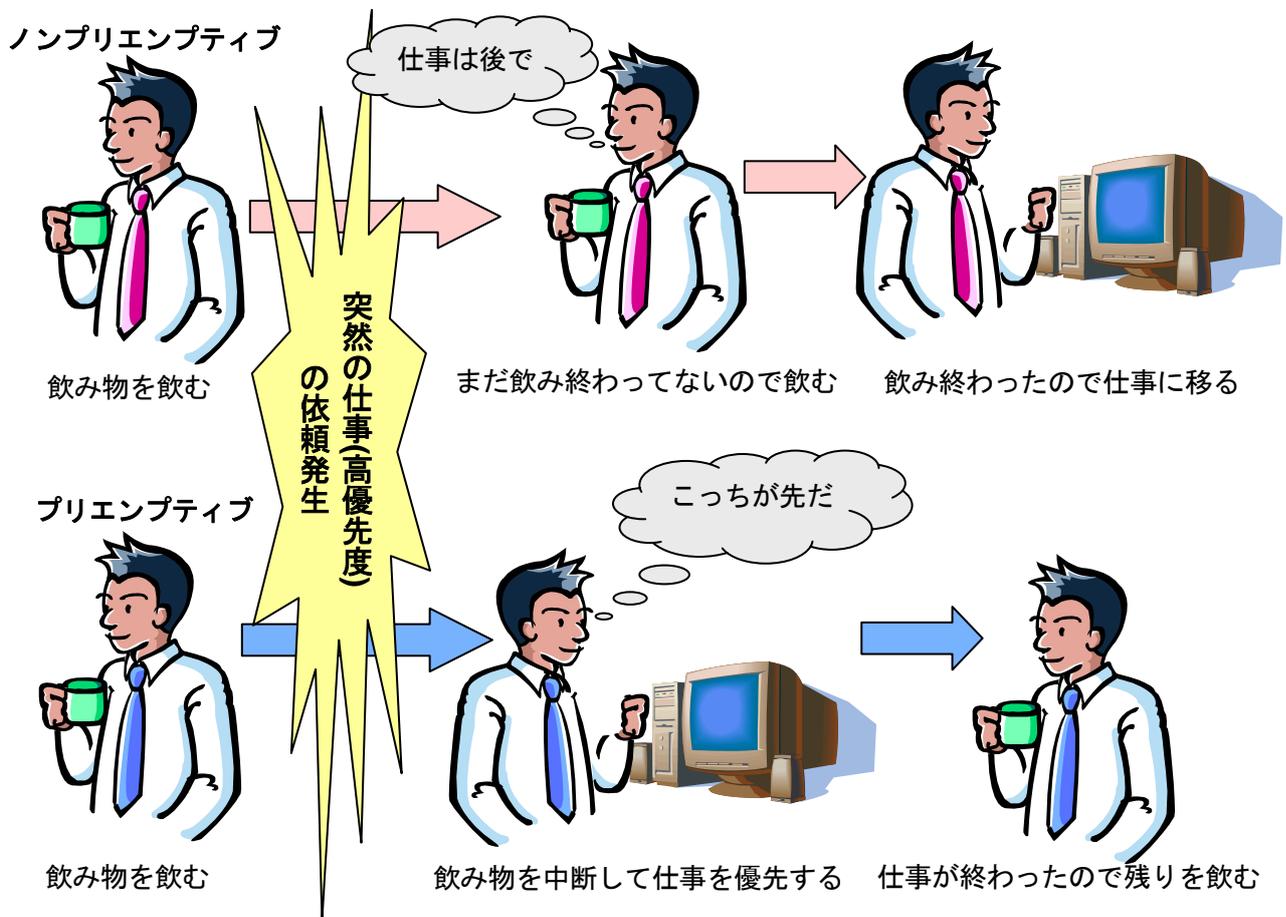



図 3.4-2 プリエンプティブとノンプリエンプティブ

それでは、実際にリアルタイム OS が並列処理を実現する仕組みを考えてみましょう。

まず、実際にリアルタイム OS の仕組みについて説明する前に、これまで「処理」と呼んでいた言葉を別の言葉に置き換えることにします。
「2.4 並行処理単位とは」で、組込み OS を分類した際に、プロセスモデルとスレッドモデルとに分類しましたが、この OS の分類方法は極端にいえば「OS がメモリをどの様に処理に割り当てるか」という分類になります。ですから、実装する OS がプロセスモデルであれば「処理=プロセス[17]」、スレッドモデルであれば「処理=スレッド」になります。
しかし、これは実装する OS を決めた際に決まる物で、上流行程（まだ OS の選定などはせず、機能や処理などを検討解析している段階）では、処理がプロセスであるのか、スレッドであるのかはわかりません。このような段階で、単に「並行処理」の単位として用いられる用語が「タスク」です。

タスクは、自身がリアルタイム OS によって切り替えられているとは知りません。タスクは、与えた処理を確実にマイコンが実行してくれていると思っていますから、リアルタイム OS は仮想的なマイコンをタスクに見せていると考えることができます。しかし、タスク側で切り替えを意識させない以上、切り替え作業に関するすべては OS が行うことになります。そのため、リアルタイム OS が個々のタスクを管理するための方法が必要となります。

管理の方法は、リアルタイム OS ごとに異なりますが、たとえば **図 3.4-3** のような情報の集合体をタスクごとに作成し、管理する方法があります。
この中で注目して欲しいのが「汎用レジスタ退避領域」です。タスクの切り替えを行う場合には、切り替えた対象となるタスクにいつ処理が戻っても、切り替わる直前まで行っていた状態から再開する必要があります。これを実現するために、実行中だった状態のプログラムカウンタ（マイコンが実行していた命令のアドレス）、スタックポインタ（現在のスタックのアドレス）、またマイコンが持つ各レジスタの値などを保存する必要が発生します。この「汎用レジスタ退避領域」は、それらレジスタの値を保存する領域になります。タスクの切り替えは、タスク状態や優先度、起動アドレスなども管理する必要がありますが、タスクとタスクを切り替える手法は、この汎用レジスタ領域の退避によって実現されているのです。

なお、汎用レジスタ退避領域に保存する「汎用レジスタ」、「スタック」などのマイコン資源を「**タスクコンテキスト**」といい、タスクの状態、タスクの優先度などを集めたタスク情報を「タスクコントロールブロック」、略して「**TCB**」といいます。リアルタイム OS は、この TCB を走査して実行するタスクを判定し切り替えます。
リアルタイム OS が、次に実行するタスクを決定する処理を「スケジューリング処理」、タスクを切り替えることを「ディスパッチ処理」といい、ディスパッチ処理を行うリアルタイム OS の機能部分を「**ディスパッチャ**」といいます [18] (**図 3.4-4**)。

図 3.4-3 TCB 構造の例

ではそのディスパッチを行う手順ですが、各タスクの TCB をタスク切り替えの際に、その都度すべての TCB を走査するのは効率的ではありません。このディスパッチの処理は、並行処理を行うための「本来の処理とは関係のない」処理ですから、できるだけ短く高速に実行できることが望まれます。そのため、リアルタイム OS は、

[17] プロセスモデルの場合は、プロセス内部に複数のスレッドが持てるものがありますが、ここではプロセスという大きなまとまりで考えて下さい。
[18] タスクを切り替えるという意味合いで、「ディスパッチ」のことを「スイッチング」と呼ぶ場合も有りますが、OSEK OS 仕様ではディスパッチと呼んでいますので、本書もこれに従います

すべてのタスクを走査するのではなく、起動要求を受けて実行可能な状態になったタスクを優先度順で整列させた状態で TCB を管理することにより、次に起動するタスクを短時間で見つけるのを可能にしています。

ここでいう実行可能状態にあるタスクとは、実行を求められている処理ですが、他の処理が実行されているためマイコンによる実行を待っているタスクのことを指します。リアルタイム OS が何故「実行可能状態」のタスクのみを走査するかは、タスクが「どのような状態になる可能性があるか」を考えれば容易に理解することができます(**図 3.4-5**)。

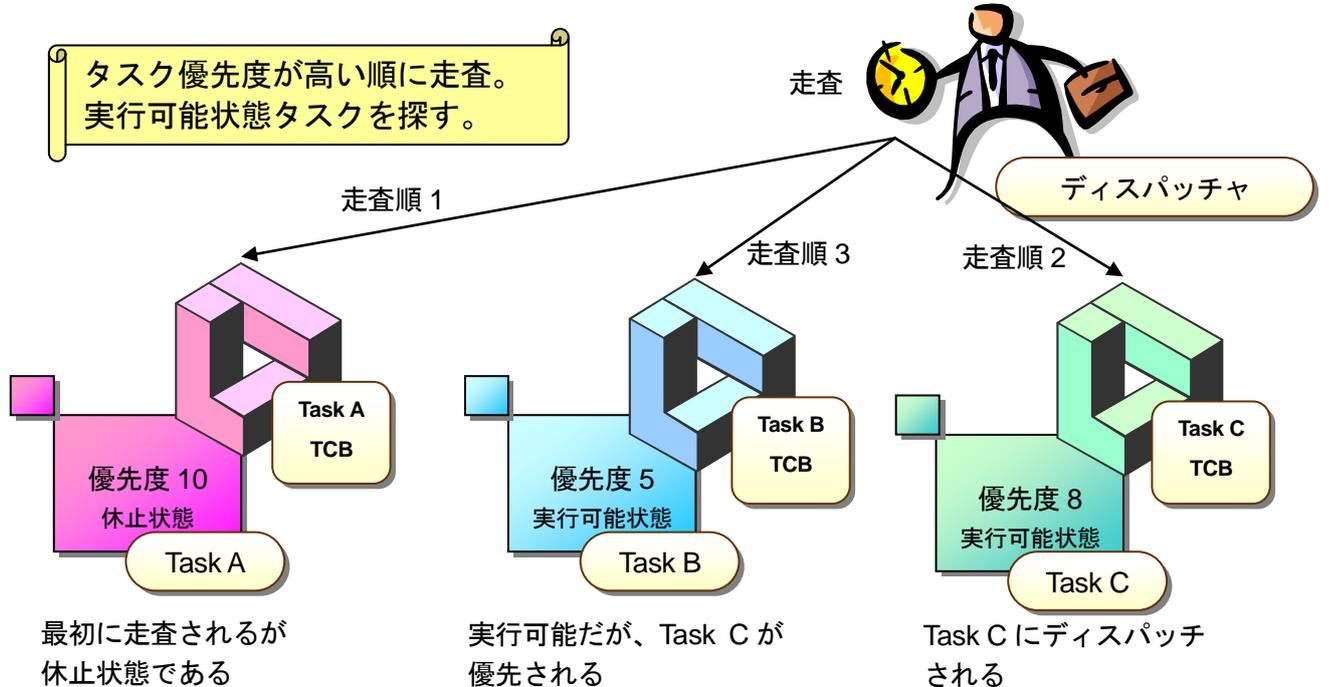



図 3.4-4 タスク管理の構造

タスクの初期状態は「休止状態」という状態です。休止状態は「実行許可」が下りていない状態で、リアルタイム OS から見れば「存在するのは知っているが、まだ関係ない」というタスクです。これに実行許可が下りると、「実行可能状態」になります。

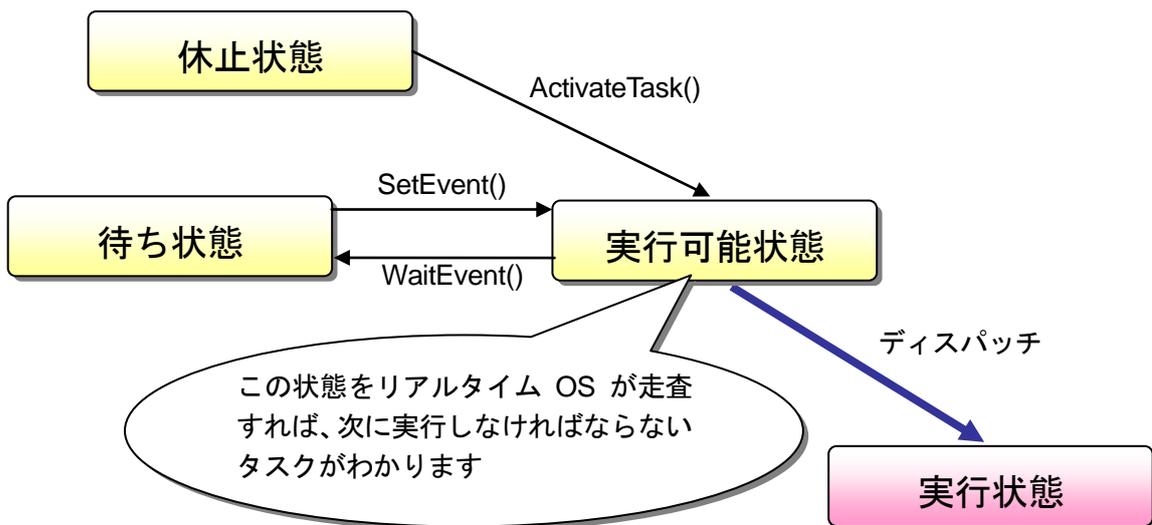



図 3.4-5 実行状態のみ監視する理由

別のタスクの処理との同期をとるため､｢今はタスクの実行を待って欲しい｣という状態になるかも知れません。その場合は、実行してもよい状態になるまで待ちます。この状態のときのタスクを「待ち状態」といいます。

もし、待つ必要が無くなり、タスクを実行できる状態になったときには、はれて「実行状態」という状態になり、タスクが実行されます。

あらためてこれだけの種類をリアルタイム OS が走査して実行すると考えると、非常に効率が悪いことがわかります。しかし、**図 3.4-5** を見ると「休止状態」や「待ち状態」のタスクは突然実行することは無く、一度「実行可能状態」になってから実行されます。このように考えると、「実行状態」になるためには一度「実行可能状態」になる必要がありますから、この状態のタスクを走査すれば良いということがわかります。

実行可能状態になっているタスクの TCB は、優先度順で並ぶと説明しました。これは「キュー」といわれるデータ構造になっています。キューは極端にいえば「ラーメン店の行列待ち」と同じ構造です。ラーメン店の行列待ちは、後ろに次々と人が並んでいきます（キューに並ぶことを「キューイング」といいます）。もちろん、人が行列の途中で入るのは「御法度」です。また、準備ができると一番はじめに並んでいた人が、一番はじめに出て行きます。これを First In First Out といい、一般に略して「FIFO」といいます。

ここで注意して欲しいのは、TCB を管理するリストはキュー構造ですが、優先度順に並んでいるという点です。つまり、すべて同じ優先順位であれば上記の「FIFO」が成り立ちますが、この TCB の場合、優先順位に合ったところに配置されます。たとえば、**優先順位の一番高いタスク**が後からやってきた場合、そのタスクは先頭に並びます。つまり、キュー構造では「御法度」な、**並び替えが発生**するのです。ラーメン店の行列待ちに当てはめて考えると、この TCB のリスト構造は「お得意様優先」なラーメン店といえます(**図 3.4-6** [19])。

このように TCB のリストを管理すれば、次に実行状態にする必要があるタスクは常に、リストの先頭に位置することになり、素早く走査が可能になります。この特性から実行可能状態タスクの状態を持つ TCB のリストを「**レディキュー**」といいます。

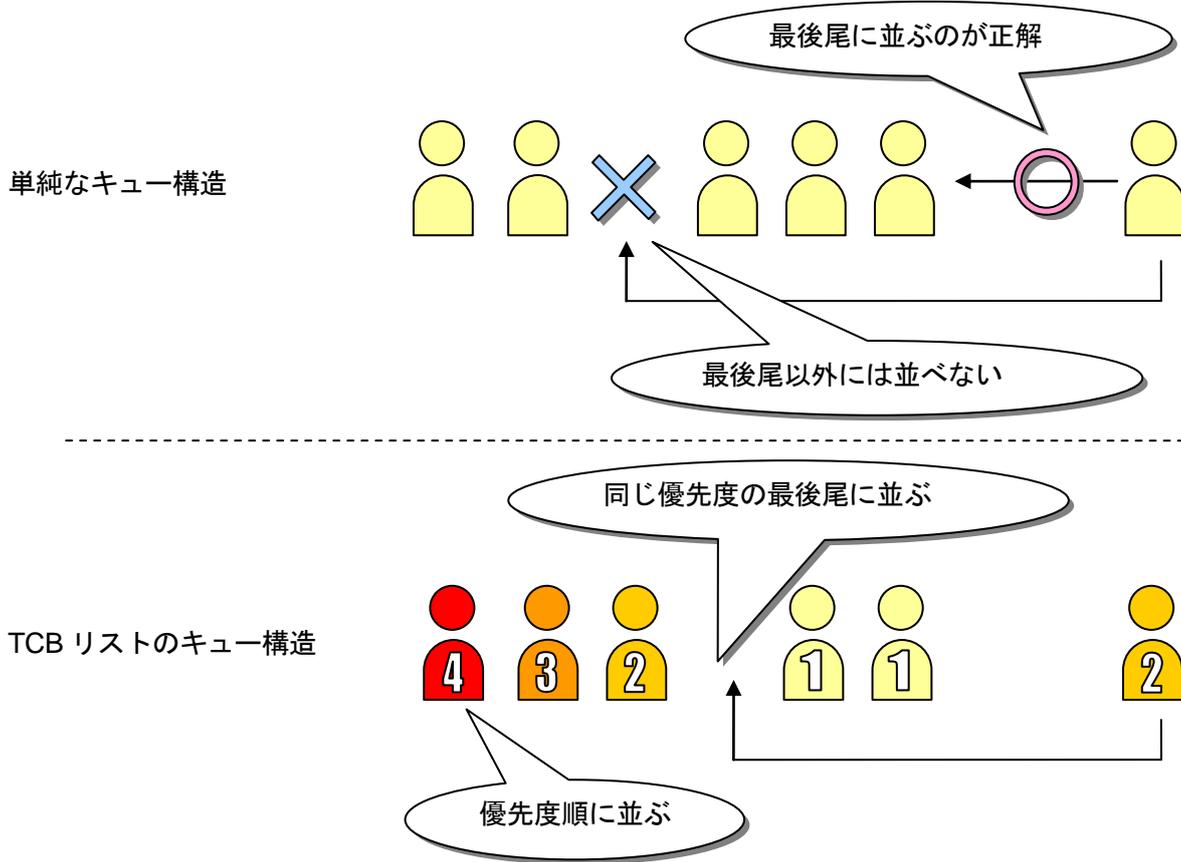



図 3.4-6 キュー構造の違い

タスクのディスパッチは、**図 3.4-7**の順番で行われます。**図 3.4-7** は、「タスクＡ」と「タスクＢ」があり、現在「実行状態」であるタスクＡが、待ち状態になるため「実行可能状態」であったタスクＢが開始されるという例です。順番に追って見ていきましょう。

[19] OSEK OS では、数値の大きいほうが高い優先度となりますので、この図もそれにあわせてキューの先頭に行くほど数値が大きくなっています。

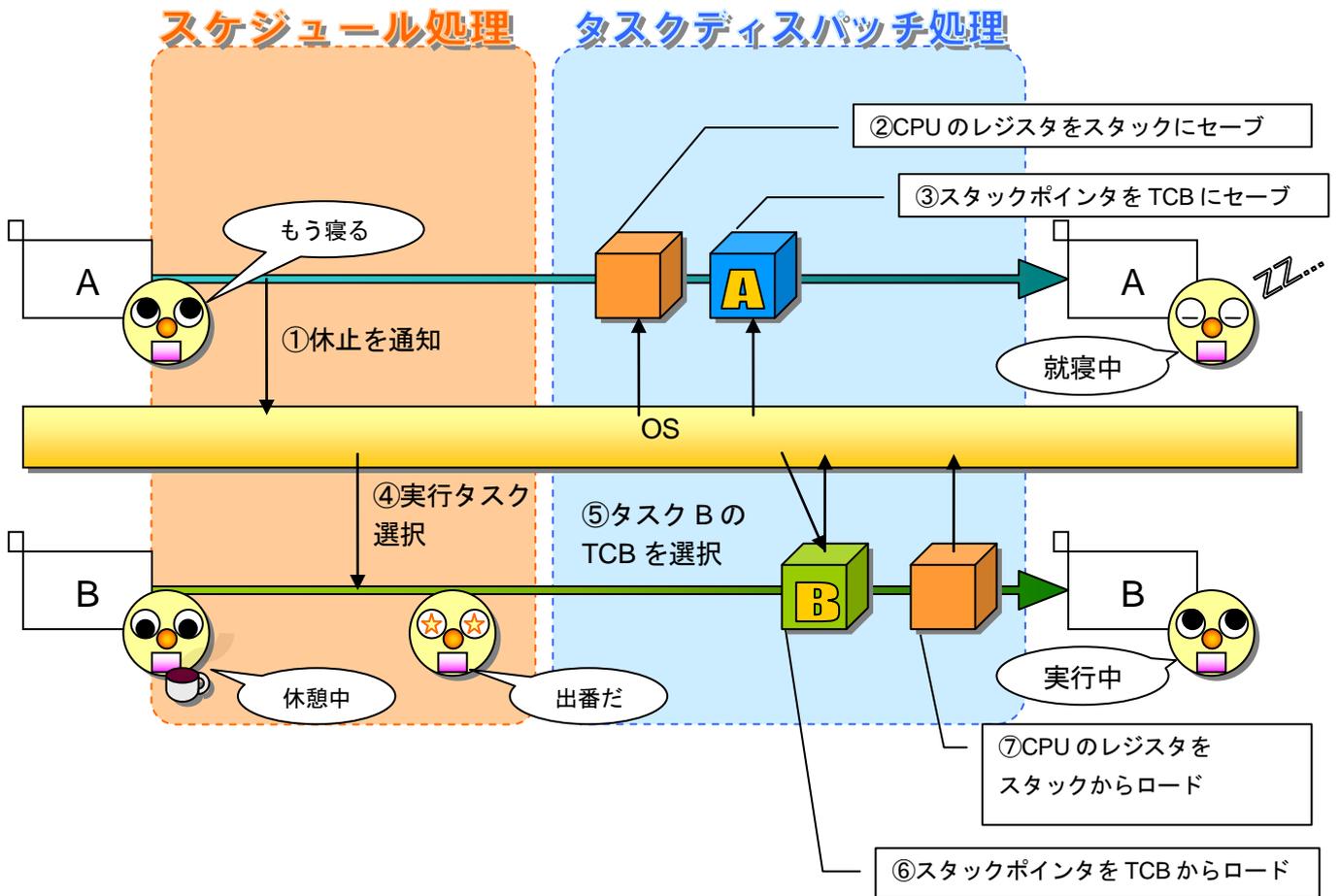



図 3.4-7 タスクディスパッチの例

① まず、タスク A は自分自身が休止状態になることを OS に通知します。
② 現在実行しているタスク A のマイコンレジスタをタスク A 用に割り当たっているスタック領域にセーブします。
③ セーブ完了後、今度はスタックポインタをタスク A の TCB にセーブします。
④ 通知された OS はスケジューリング処理を行い、次に実行するタスク B を選択します。
⑤ 次に実行するタスク B を選択できた場合、OS は管理している TCB をタスク A からタスク B に変更します。
⑥ 変更後、スタックポインタをタスク B の TCB からロードします。
⑦ ロード後、タスク B のスタックからマイコンのレジスタをロードします。

「タスクディスパッチ処理」は、次の実行タスクにタスク B が選択された後からの処理になります。現在実行しているタスクの状態を、スタックに保存して保存したスタックポインタを TCB に保存しておきます。次に、タスク B の TCB を呼びだし、TCB のスタックに保存されているスタックポインタを使って、前回処理されていたところまでの状態に回復します。これで、マイコンの実行はタスク B に移り、ディスパッチ処理が完了したことになります。

3.4.2 コンフォーマンスクラス

OSEK OS ではプロセッサやアプリケーションに対するスケーラビリティを確保するため、OS の機能セットとして 4 種類(BCC1/BCC2/ECC1/ECC2)のコンフォーマンスクラスが定義されています。コンフォーマンスクラスごとに次のような違いがあります。

表 3.4-1 コンフォーマンスクラスの種類

<table>
<tr><th>項目</th><th>BCC1</th><th>BCC2</th><th>ECC1</th><th>ECC2</th></tr>
<tr><td>タスクの種類</td><td>BT</td><td>BT</td><td>BT/ET</td><td>BT/ET</td></tr>
<tr><td>多重要求</td><td>×</td><td>○</td><td>×</td><td>○
ET にも対応</td></tr>
<tr><td>休止状態でない
タスク最大数</td><td colspan="2">８</td><td colspan="2">２５５
※BT/ET の合計数</td></tr>
<tr><td>１優先度あたりのタスク数</td><td>１</td><td>複数</td><td>１</td><td>複数</td></tr>
<tr><td>タスクの
最大イベント数</td><td colspan="2">―</td><td colspan="2">３２</td></tr>
<tr><td>優先度数</td><td colspan="2">８</td><td colspan="2">１６</td></tr>
<tr><td>リソース</td><td>RES_SCHDULER</td><td colspan="3">２５５(RES_SCHDULER 含む)</td></tr>
<tr><td>内部リソース数</td><td colspan="4">2</td></tr>
<tr><td>アラーム数</td><td colspan="4">２５５</td></tr>
<tr><td>アプリケーションモード</td><td colspan="4">８</td></tr>
</table>

RES_SCHDULER･･･スケジューラを指す

BT は基本タスク、ET は拡張タスクを示します。ＢＴは待ち状態がないタスクですが、その分リソースの消費が抑えられます。ET は待ち状態のあるタスクですが、その分基本タスクに比べてリソースを消費します。

表 3.4-1から上位レベルは下位レベルを網羅できることがわかります。たとえば ECC2 は ECC1 のコンフォーマンスを網羅することができますが、ECC1 は、ECC2 のコンフォーマンスを網羅できません。

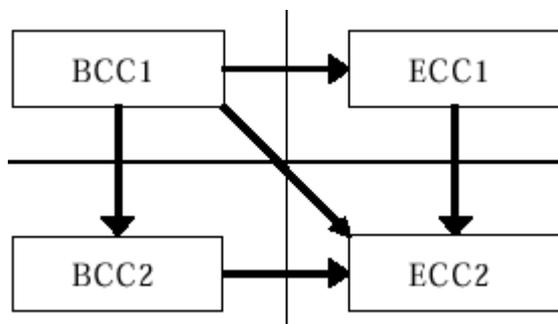



図 3.4-8 コンフォーマンスクラス関連図

3.4.3 スケジューリング方式

OSEK OS にはフルプリエンプティブスケジュール、ノンプリエンプティブスケジュール、ミクスドプリエンプティブスケジュールの 3 つのスケジューリング方式があり、以下のような特徴があります。

(i)フルプリエンプティブ

優先度の高いタスクが優先されるスケジューリング方式です。高い優先度のタスクが実行可能になれば再スケジュールされます。

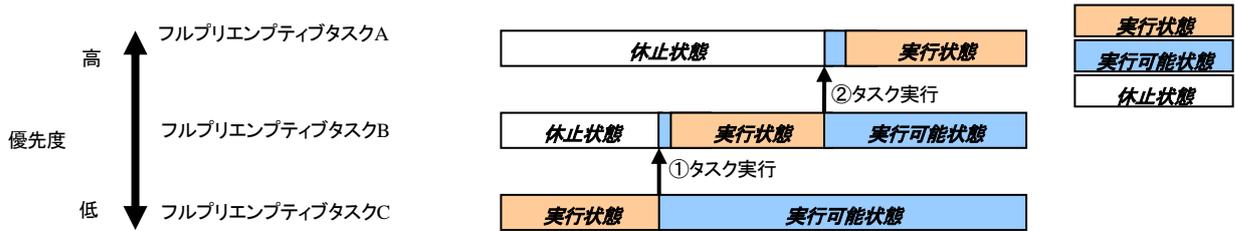



図 3.4-9 フルプリエンプティブ の例

① タスク C がタスク B を実行要求します。タスク C はフルプリエンプティブ のため、優先度が高いタスク B が実行状態になり、タスク C は実行可能状態になります。

② タスク B がタスク A を実行要求すると、①と同様に優先度の高いタスク A が実行状態になり、タスク B は実行可能状態になります。

(ii) ノンプリエンプティブ

現在動作しているタスクが優先されるスケジューリング方式です。再スケジューリングを行うシステムコールを呼ぶと再スケジュールされます。

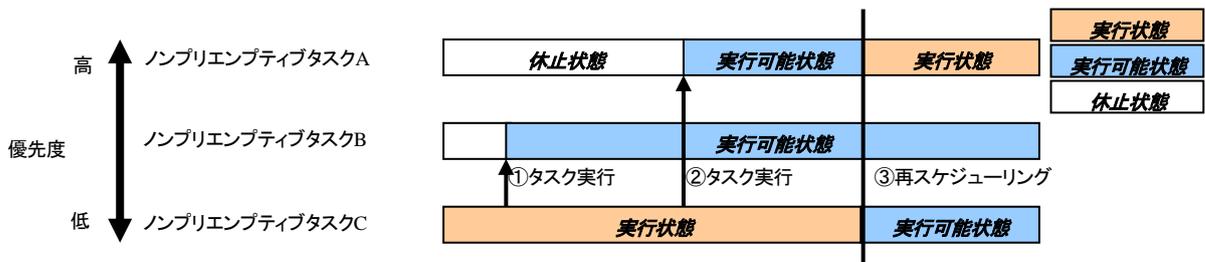



図 3.4-10 ノンプリエンプティブ の例

① タスク C がタスク B を実行要求します。タスク C がノンプリエンプティブ の場合、タスク B は実行可能状態になります。タスク B のほうが高優先度ですが元々実行状態であったタスク C が実行状態を継続します。

② タスク C がタスク A を実行要求します。①と同様に高優先度のタスク A は実行可能状態になりますが、タスク C は実行状態を継続します。

③ タスク C が再スケジューリング要求を行うと、この段階で一番優先度の高く実行可能状態であったタスク A が実行状態になり、タスク C は実行可能状態になります。

(iii) ミクスドプリエンプティブ

上記２つの特性をタスクごとに設定することができるスケジューリング方式です。

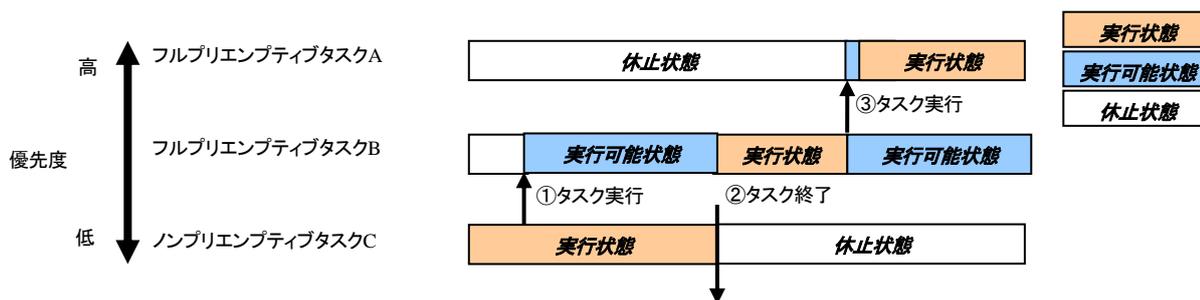



図 3.4-11 ミクスドプリエンプティブ の例

① タスク C がタスク B を実行要求します。タスク C がノンプリエンプティブ の場合、タスク B は実行可能状態になります。タスク B のほうが高優先度ですが元々実行状態であったタスク C が実行状態を継続します。

② タスク C が自分自身を終了すると、タスク C は休止状態になり、実行可能状態であったタスク B が実行状態になります。

③ タスク B がタスク A を実行要求します。タスク B はフルプリエンプティブ のため、優先度が高いタスク A が実行状態になり、タスク B は実行可能状態になります。

3.4.4 イベント

タスク-タスク間、タスク-ISR 間で同期を行うための機能としてイベント機能があります。イベント機能を使用することで、ある状態になるまで処理を待ち、その状態になったら処理を再開することができます。尚、イベントの設定及び取得は全てのタスクで実行可能ですが、イベント待ち及びクリアを実行できるのは拡張タスクのみで基本タスクは実行できないため注意が必要です。

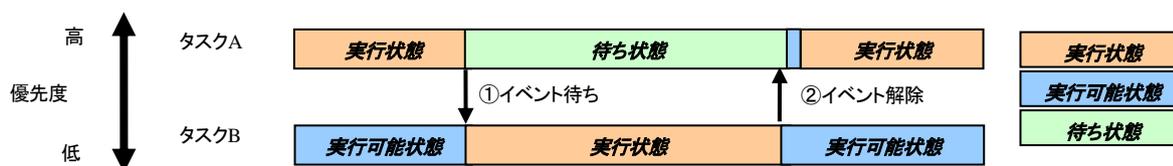



図 3.4-12 イベントの例

① イベント待ちするとタスク A の待ち状態になり、実行可能状態のタスク B が実行可能状態になります。

② タスク B がイベントを解除します。イベント待ちしていたタスク A はタスク B より優先度が高いためタスク A が実行状態になり、タスク B は実行可能状態になります。

3.4.5 アラーム

アラーム機能は要求した時間になると通知をくれる機能です。アラーム機能を使用することで、ある時間になったときにタスクを起動するなど特定の処理を実行することができます。アラーム機能の詳細は**6.3 アラーム機能** を参照して下さい。

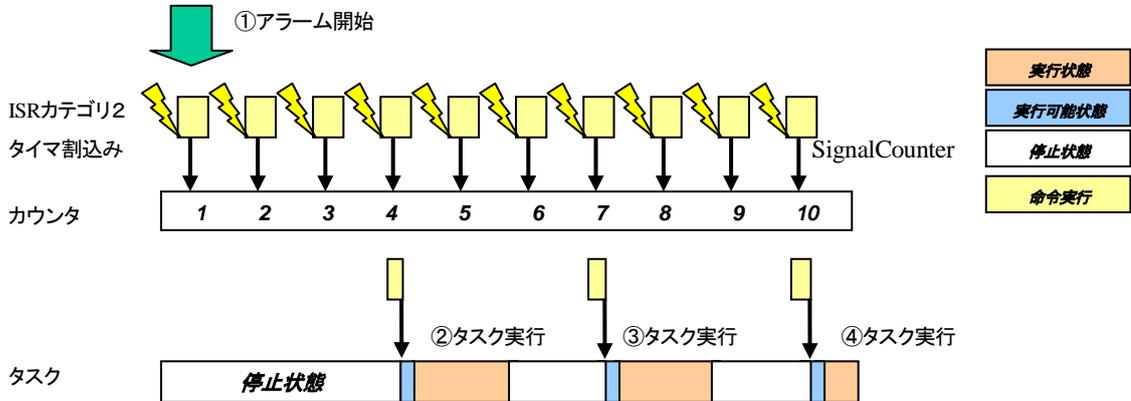



図 3.4-13 アラームの例

図 3.4-13は、アラームの初期起動を 4、周期を 3 に設定し、そのときにタスクを起動する設定にした場合の動作を示しています。

① カウンタが 1 のときにアラームを開始します。カウンタはタイマ割込みにより一定間隔でカウントアップされていきます。
② カウンタがアラームの初期起動値 4 に到達すると、停止状態のタスクが実行状態になります。その後タスクは自分自身を終了し停止状態になります。
③ カウンタが②から周期値 3 カウントアップし 7 に到達すると、停止状態のタスクが実行状態になります。その後タスクは自分自身を終了し停止状態になります。
④ カウンタが③から周期値 3 カウントアップし 10 に到達すると、停止状態のタスクが実行状態になります。その後タスクは自分自身を終了し停止状態になります。以降、カウンタが 3 カウントアップする度に同様の動作を繰返します。

3.4.6 リソース

共有資源（リソース）の確保を調停するための機能としてリソース管理機能があります。OSEK OS ではリソースを確保することで優先度を一時的に上げる優先度上限プロトコルによりリソース管理機能を実現しています。リソース管理機能の詳細は**6.4 排他制御** を参照して下さい。

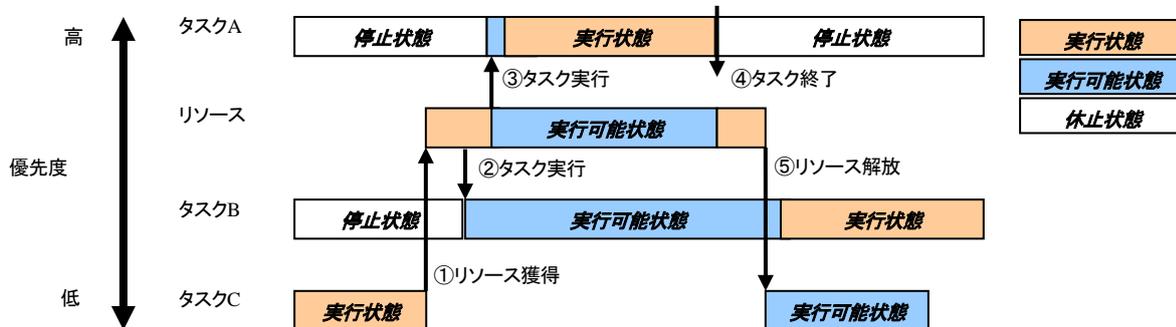



図 3.4-14 リソースの例

① リソースを獲得するとタスク C の優先度はリソースの優先度まで上がります。
② タスク C からタスク B を実行要求します。リソースを獲得しているタスク C はタスク B より優先度が高いため実行状態を継続します。タスク B は実行可能状態になります。
③ タスク C からタスク A を実行要求します。タスク C はリソースを獲得していますが、タスク A のほうがリソースより優先度が高いためタスク A が実行状態になり、タスク B は実行可能状態になります。
④ タスク A が自分自身を終了すると実行可能状態の中で一番優先度の高いタスク C が実行状態になります。
⑤ タスク C がリソースを解放するとタスク B はタスク C より優先度が高くなるためタスク B が実行可能状態になり、タスク C が実行可能状態になります。

3.4.7 フックルーチン

OSEK OS では特定のタイミングで OS から特定のルーチンを実行する仕組みとしてフックルーチンがあり、スタートアップフック、シャットダウンフック、プレタスクフック、ポストタスクフック、エラーフックの 5 つのフックルーチンが提供されています。

表 3.4-2 フックルーチンの種類

フックルーチン	実行タイミング
スタートアップフック	OS 起動時
シャットダウンフック	OS 終了時
プレタスクフック	ディスパッチ直前
ポストタスクフック	ディスパッチ直後
エラーフック	エラー発生時

以下に、スタートアップフックの例を示します。

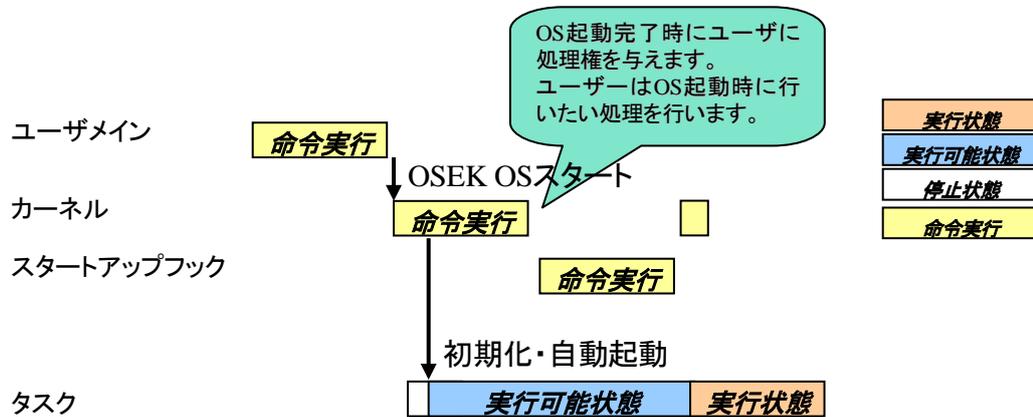



図 3.4-15 スタートアップフックの例

4 リアルタイム OS を使用した組込み開発手法

4.1 組込みシステム開発

4.1.1 組込みシステム開発の流れ

実際の組込みシステム開発では、ドキュメントを作成する必要があります。ドキュメントは第三者との情報共有のために行い、第三者が見ても分かるようにする必要があります。

製品開発を例にした場合、ドキュメントは「機能仕様書」「設計仕様書」「テスト仕様書」など様々な種類を作成します。

機能仕様書とは、製品を使用するユーザの立場から考えたシステムに求める機能を書いたものです。

設計仕様書とは、機能仕様書に記述された機能をどの様に実現するかを記述したシステム提案書といえます。

テスト仕様書とは、機能仕様書や設計仕様書に記述された内容通りに実装されているかを確認するための、テスト項目やテスト手順を書いたものです。

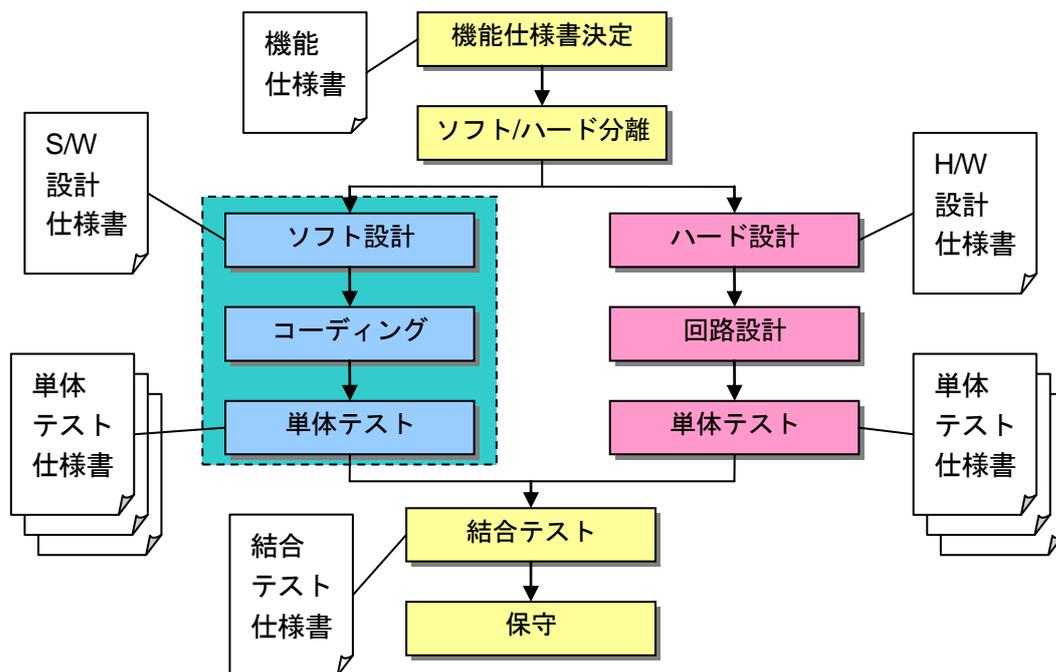



図 4.1-1 組込みシステムの開発フロー

4.1.2 クロス開発環境

組込みシステムのソフトウェアは、一般のパッケージソフト（PC 上で動作するアプリケーションの作成等）の開発方法と異なり、「**クロス開発環境**」を用います。

パッケージアプリケーションの場合、開発環境と実行環境の開発環境が同一の環境下で行われます。もし、マイコンや、メモリ等の違いが PC にあっても、OS がそのハードウェアの差を吸収してくれるため、パッケージ

アプリケーションの開発者は、それらの違いを意識することなく、開発することができます。
　これに対して、組込み開発環境の場合、開発環境と実行環境の違いは大きく、使用するマイコンや、メモリの容量、またそれを吸収してくれる OS の違い（実行環境に OS がない場合もある）など、パッケージアプリケーションの開発環境とは大きく異なります。
　開発環境と実行環境を同一の環境下で開発することを「セルフ開発環境」と呼びます(**図 4.1-2**)。対して、開発環境と実行環境が異なる環境で開発されることを「**クロス開発環境**」と呼びます(**図 4.1-3**)。また、クロス開発環境の場合、開発環境に使われるコンピュータを「**ホスト**」と呼び、開発対象となるコンピュータを「**ターゲット**」と呼びます [20]。クロス開発環境を実現するには、「**クロスツール**」[21]が必要になります。

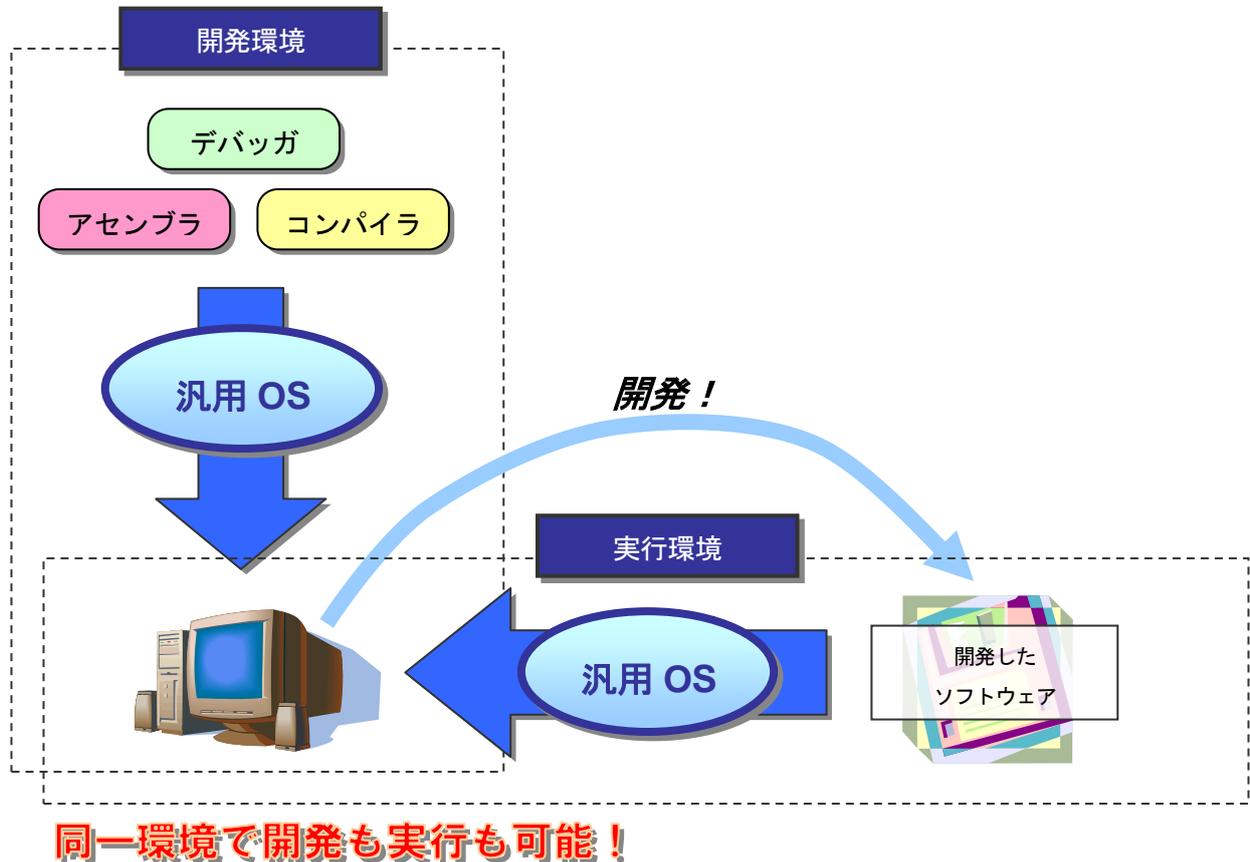



図 4.1-2 セルフ開発環境

[20] システムによって、ホストコンピュータ、ホストシステム、ターゲットボード、ターゲットシステムと、呼び名が変わりますが、本書では単に「ホスト」と「ターゲット」と呼びます。
[21] ここで言う「クロスツール」は、プログラマが高等言語で作成したソースコードを解釈し、開発に使用しているのとは異なる機種で実行可能な機械語のプログラムを生成するソフトウェアのこと。プリプロセッサ、コンパイラ、アセンブラ、リンカ、ライブラリアン等を含んでいます。

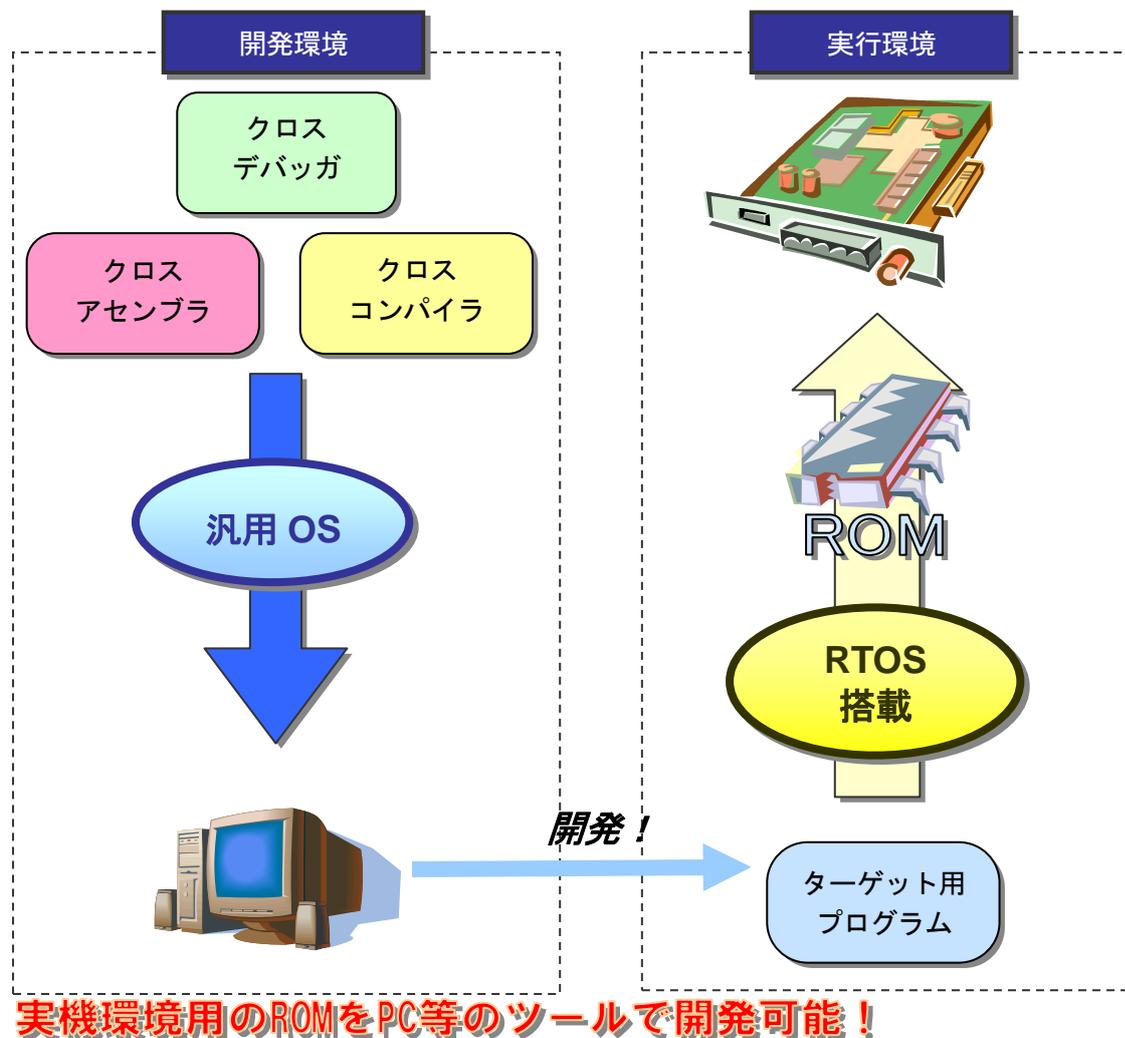



図 4.1-3 クロス開発環境

4.1.3 ROM 化

クロスツールを用いてホスト上で作成するオブジェクトはマイコン用のものになります。マイコン用の命令コードに変換されているため、命令コードが異なるホスト側ではこれを実行することができません。

ホスト側で生成したオブジェクトファイルをマイコン上で実行するためには、プログラムコードを FLASH ROM や UVEEPROM などの ROM に書き込む「ROM 化」という作業を行い、ターゲットに実装します。プログラムに問題がなければ、ターゲット上で作成したオブジェクトが実行されます。

しかし、生成したオブジェクトはそのままの形では ROM に書き込むことはできません。ROM に書き込むためには、プログラムコードを ROM に書き込むためのフォーマットに直さなければいけません。ROM に書き込むためのフォーマットで書かれたファイルを HEX ファイルといいます。

HEX ファイルにはどのアドレスに何の値を書き込むかという情報が書かれます。HEX ファイルのフォーマットとしては主に

- インテル HEX 形式
- モトローラ S レコード形式

という２つがあります。これらのファイルを ROM ライタで読み込むことで ROM 化が行われます。

これらの流れをまとめると、**図 4.1-4**に示すようになります。

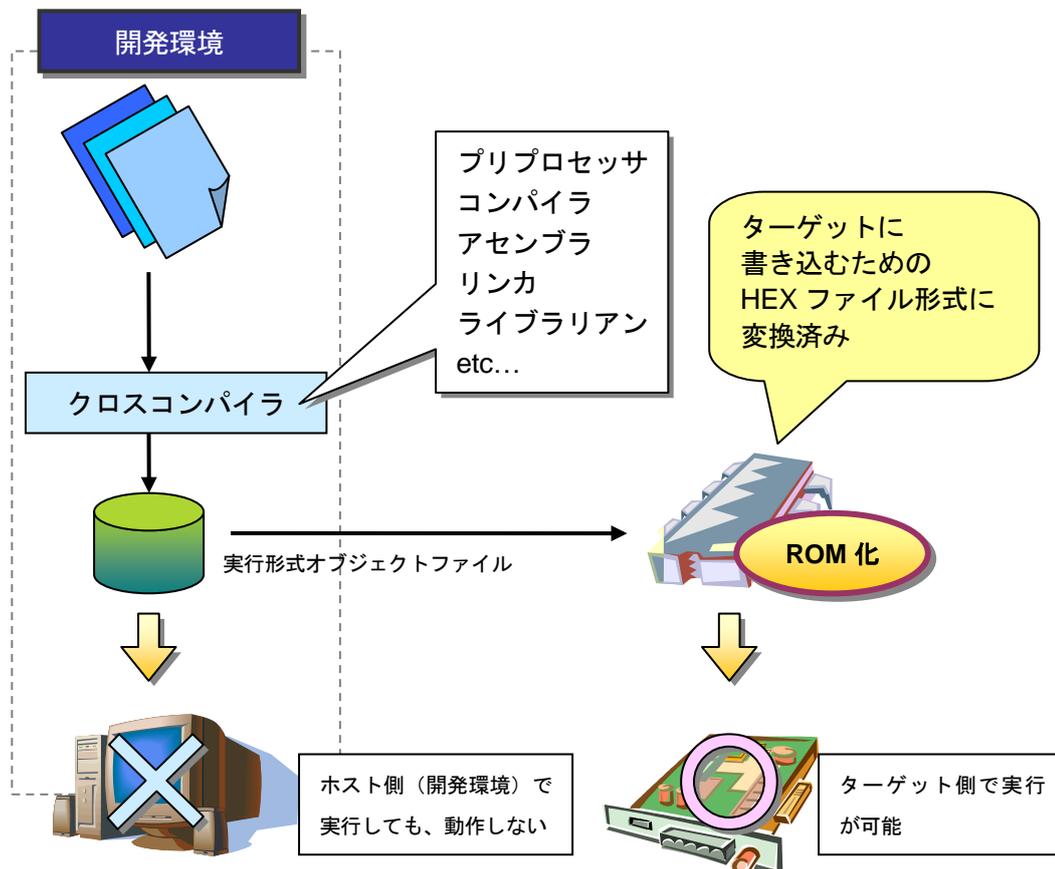



図 4.1-4 ROM 化の流れ

4.2 リアルタイム OS の有無による開発の違い

4.2.1 リアルタイム OS を使用しない場合の開発

リアルタイム OS を使用しない場合の開発の流れは**図 4.2-1**のようになります。

クロス開発環境においては、**図 4.2-1**のようなコンパイラ、アセンブラ、リンカ、HEX ファイル書込み、ROM 書込みという流れが基本になります。流れを 1 つずつ見てみましょう。

C ソースで作成してコンパイルし、アセンブラソースを生成します。ユーザが作成するアプリなどが一般的には C ソースで記述されます。

C ソースから生成されたアセンブラソースと、はじめからアセンブラ言語で記述されたアセンブラソースを一緒にアセンブルし、オブジェクトファイルを生成します。RAM 領域の初期化や、はじめに処理する関数の呼出しなどの初期処理（スタートアップルーチン）はアセンブラソースで直接記述されます。

生成したオブジェクトファイルと、必要に応じたライブラリファイルをリンカにより結合します。結合されたファイルは、実行するための情報が揃った結合オブジェクトファイルとなります。

ここまでできたら、残りは**4.1.3 ROM 化**でも説明した ROM 化を行い、マイコン上にプログラムを書込みます。

問題がなければマイコン上で開発したプログラムが動作し、一通りの開発は完了となります。

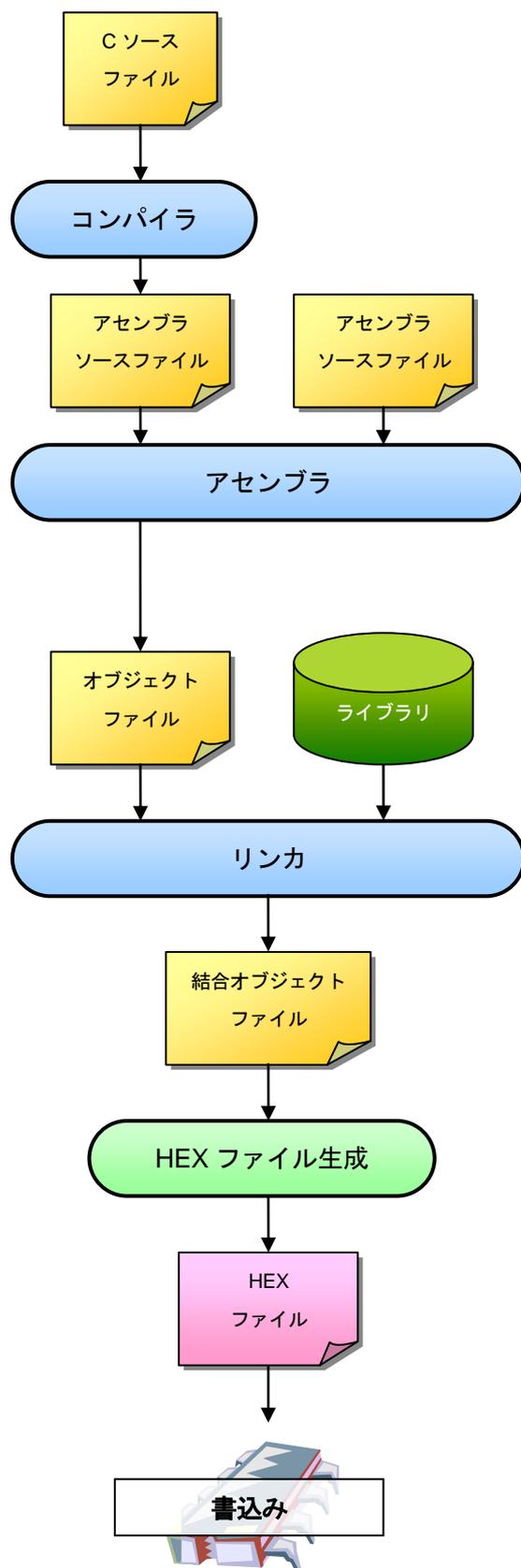



図 4.2-1 TOPPERS Automotive Kernel 未使用時の HEX ファイルができる流れ（ROM 化プログラム）

4.2.2 リアルタイム OS を使用した場合の開発

リアルタイム OS を使用した場合の開発の流れは**図 4.2-2**のようになります。

リアルタイム OS 使用時の場合でも、**図 4.2-1**で示した流れがベースになっており、一部情報が追加された形になります。

リアルタイム OS を使用しない場合と比較しながら、流れを１つずつ見てみましょう。

コンパイルする情報がリアルタイム OS 用に追加されています。OIL と呼ばれるカーネル情報を記述したファイルを、システム・ジェネレータ（SG）と呼ばれるツールでＣソースファイルに変換します。ユーザプログラムを記述したＣソースと、SG により生成されたＣソースを一緒にコンパイルし、アセンブラソースを生成します。OIL と SG に関する詳細は**4.2.3 コンフィギュレーション** を参照して下さい。

アセンブラに関しては、リアルタイム OS を使用しない場合と同じく、Ｃソースから生成されたアセンブラソースと、初期処理を記述したアセンブラソースを一緒にアセンブルし、オブジェクトファイルを生成します。

リンクする際には、ユーザが特別必要とするライブラリ（**図 4.2-2**の緑色で示したライブラリ）の他に、リアルタイム OS を使用する上で必要になるカーネルライブラリも一緒にリンクします。これにより、リアルタイム OS を使用するための情報を含めた結合オブジェクトファイルが生成されます。

必要なオブジェクトファイルが生成されたら、ここからはリアルタイム OS を使用しない場合と同じで、ROM 化を行い、マイコン上にプログラムを書込みます。

これにより、リアルタイム OS を実装したプログラムがマイコン上で動作します。

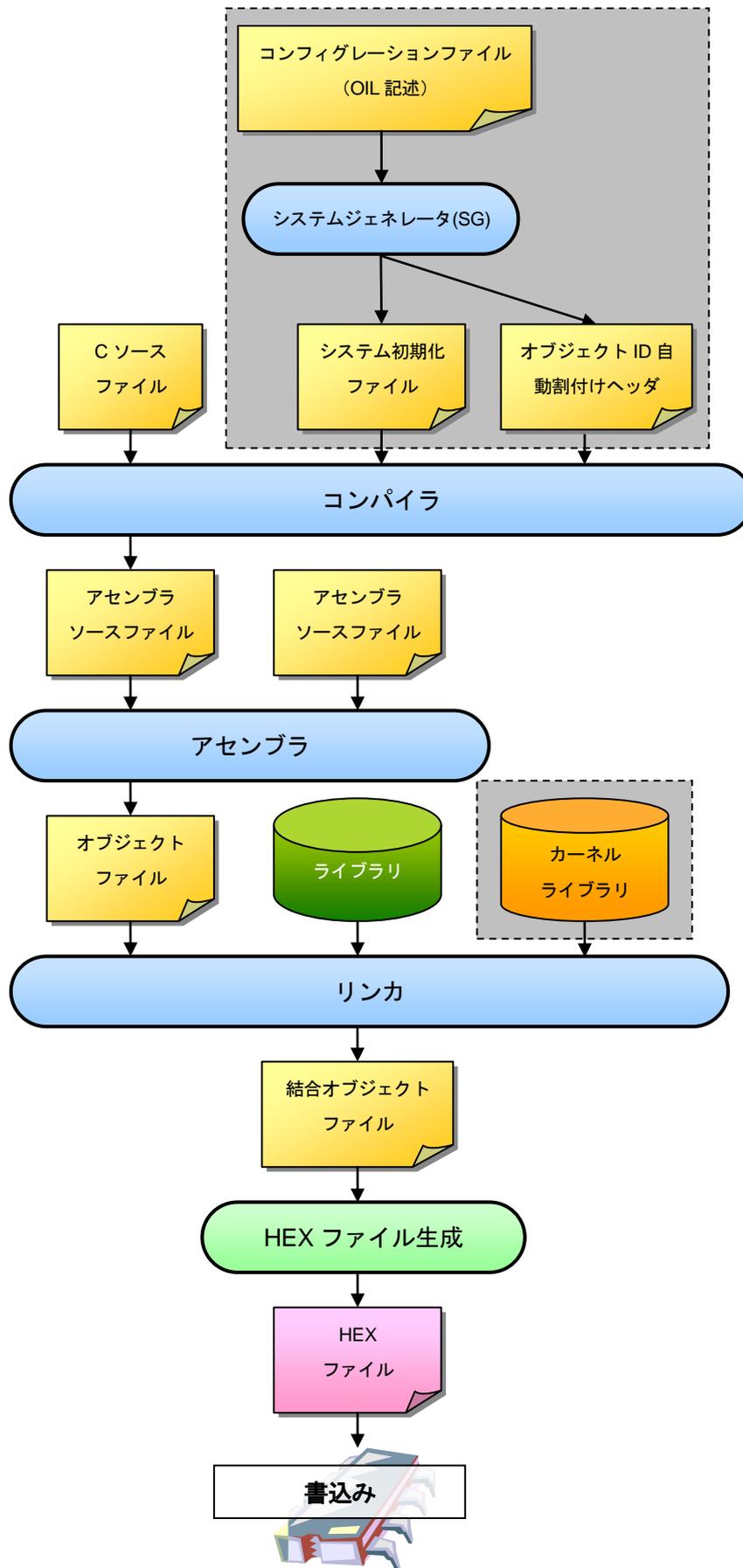



図 4.2-2 TOPPERS Automotive Kernel 使用時の HEX ファイルができる流れ（ROM 化プログラム）

4.2.3 コンフィギュレーション

コンフィギュレーションとは、システムに必要なタスクやリソースなどの情報を「**静的**」に生成することです。このようなタスクやリソースなどの情報を記述したファイルをコンフィギュレーションファイルといい、このファイルをコンフィギュレータというツールに通すことで、システムの情報が静的に出力されます。

「静的」というのは、プログラムを実行する前から既に情報が用意されている場合のことをいいます。逆にプログラム実行中に情報が生成される場合を「動的」といいます。コンフィギュレーションによってできる情報は、マイコンに書かれる前の段階で定義される情報なので静的といえます。

Windows などではユーザがどれだけアプリを起動するかが不明であるため、汎用性を持たせるためにメモリ確保は動的に行う必要があります。しかし、OSEK やμITRON などのリアルタイム OS では、機能を特化しているためにあらかじめ必要なタスクやリソースが明確であり、そのために必要なメモリ量も明らかになっているため、静的にメモリ確保しておくことが可能です。

リアルタイム OS を用いた開発では、リアルタイム性を保つために無駄な処理時間を極力省く必要があり、静的に出来る処理はできる限り静的に行っておくことが求められます。そのため、リアルタイム OS を用いた開発ではコンフィギュレーションを行うことが必要になります。

4.2.3.1 システム・ジェネレータ（SG）・OIL 記述

システム・ジェネレータ（SG）とは、OSEK OS 開発で用いるコンフィギュレータのことです。OIL(OSEK Implementation Language)という OSEK 専用のコンフィギュレーションファイル記述方法で書かれたコンフィギュレーションファイルを通すことでカーネルの内部情報を生成します。

システムジェネレータと OIL の記述方法の詳細については、**5 TOPPERS Automotive Kernel の使用方法**で説明します。

4.3 デバッグ手法

4.3.1 シミュレータ

シミュレータは、実機にプログラムを実装する前の段階でプログラムを検証する方法です。シミュレータを用いた場合の一番のメリットは、実機では発見しにくい偶然発生する問題を仮想的に再現し、検証を行うことができることです。OSEK/VDX などのマルチプログラミング環境では、各タスクが処理として機能しているか、また論理的にプログラムが動作しているか（決められた設計通りタスクが動作しているか）を検証することができます。

作成したプログラムの論理検証は、シミュレータの使用が効果的です。論理検証に実機を用いる（実装、ICE（In-circuit emulator）を含むデバッグなどの場合）と、タイミングをはじめとするハードウェアの影響を受けるため、発生している問題が優先度やタスク制御などのアルゴリズムの問題なのかが明確にわかりません。そのため、プログラム完成後すぐに「プログラムを ROM に実装する」、「デバッガを用いてデバッグを行う」ということは避け、まずシミュレータで自分が作成したプログラムが正しく実行されているか確認すると、後工程の実機デバッグ作業が楽になります。

たとえば、**5章**以降で使用する開発環境「High-performance Embedded Workshop（HEW）」には、OSEK OS のようなリアルタイム OS 向けのシミュレータ機能の１つとして、「タスクトレースウィンドウ」(**図 4.3-1**)というタスク実行履歴を計測し、グラフィカルに表示する機能があります。これによって、タスクが設計どおりの動きをしているかどうかを知ることができます。

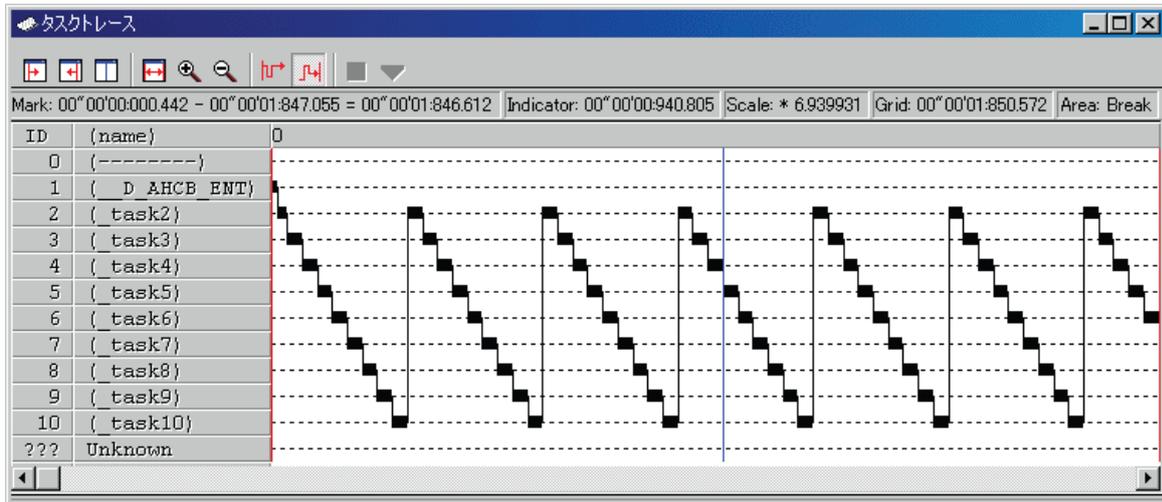



図 4.3-1 タスクトレースウィンドウ

5 TOPPERS Automotive Kernel の使用方法

本書では、下記の環境を想定して TOPPERS Automotive Kernel の使用方法について説明します。

表 5-1 TOPPERS Automotive Kernel 実装環境

種類	製品名	メーカ名
統合開発環境	High-performance Embedded Workshop	株式会社ルネサステクノロジ
開発環境(コンパイラ等)	M3T-NC308WA	株式会社ルネサステクノロジ
実装対象ハードウェア	TOPPERS Platform ボード	株式会社ヴィッツ 株式会社サニー技研 （共同開発）
リアルタイム OS	TOPPERS Automotive Kernel	TOPPERS プロジェクト
書込みソフトウェア	E8a エミュレータ	株式会社ルネサステクノロジ
シミュレータ	M32C シミュレータデバッガ	株式会社ルネサステクノロジ

移植先のハードウェア（ターゲットと呼ばれる）のハードウェア一般仕様及び通信仕様を下記に示します。

表 5-2 TOPPERS Platform ボードのハードウェア一般仕様

No.	項目	仕様
1	製品名	TOPPERS Platform ボード
2	製品型番	S810-TPF-85
3	搭載 MCU	M30855FJGP 内蔵RAM ：24K ROM ：512K SFR 領域：１K
4	メインクロック	8MHz（XIN）
5	SRAM	128KByte×２ （M5M51008DVP-55H ルネサス）
6	電源電圧	5V（USB より供給）
7	消費電力	最大 約100mA（メモリーカード非動作時）
8	外形寸法	約 150mm（W） ×110mm（D） ×45mm（H）
9	重量	約 120g

表 5-3 TOPPERS Platform ボードのハードウェア通信仕様

No.	項目	仕様
1	CAN	×１ch トランシーバ：HA13721FPK（ルネサス）
2	LIN	×1ch トランシーバ：TJA1020（Philips）
3	Ethernet	10BASET×１ コントローラ：RTL8019AS（RealTek）
4	SD/MMC カード	スロット×１（SPI モードによるアクセス）

5	USB	×1 （Full Speed 対応） ドライバ：FT232R（FTDI）

5.1 ファイルの種類とディレクトリ構成

5.1.1 ディレクトリ構成

TOPPERS Automotive Kernel のディレクトリ構成を以下に示します。

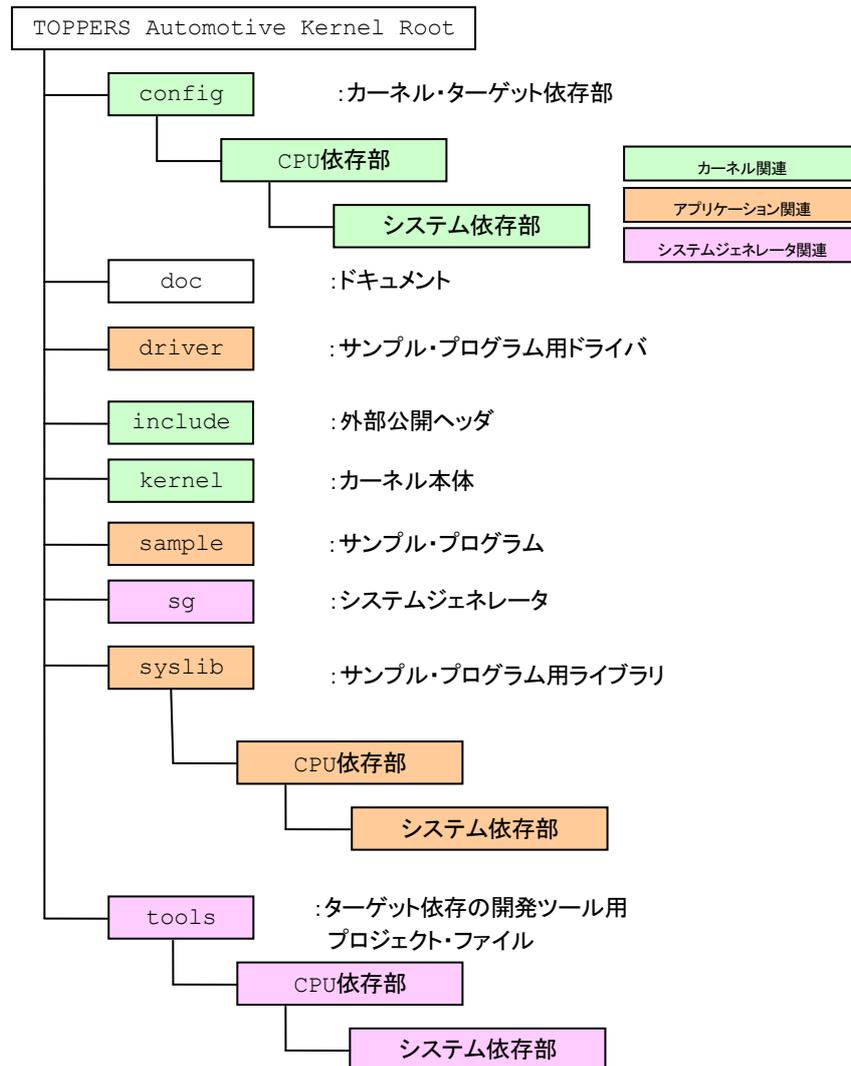



図 5.1-1 TOPPERS Automotive Kernel ディレクトリ階層

5.1.2 カーネル

TOPPERS Automotive Kernel は大きく共通部とターゲット依存部に分けられます。共通部とターゲット依存部は、CPU アーキテクチャや動作環境に関係なく実装できるかどうかで切り分けられています。CPU アーキテクチャや動作環境に依存しないほうが共通部で、カーネル全体の 80 ～ 90%を占めています。CPU アーキテクチャや動作環境に依存するほうがターゲット依存部で、ディスパッチャ、割込み制御、割込みレベル操作、スタートアップ、CPU やシステム、ツールに依存した定義や処理が主な機能です。他のターゲットに移植する場合はターゲット依存部のファイルを修正します。

5.1.3 アプリケーション

アプリケーションは特定の目的の制御を行ためにに設計されたソフトウェアです。ユーザは OS 上で動作させる処理をファイルに記述します。アプリケーションを記述する際もカーネル同様に、共通部とターゲット依存部を分けて作成すると他のターゲットへの移植作業を容易にすることができます。

5.1.4 システムジェネレータ

システムジェネレータ（SG）は OIL（OSEK Implementation Language）と呼ばれる専門の記述言語によって書かれたソーステキスト（OIL ファイル）を読み込み、カーネルが必要とする C 言語のソースファイル及びその関連ファイルを生成するツールです。
ユーザは OIL ファイルを準備する必要があります。

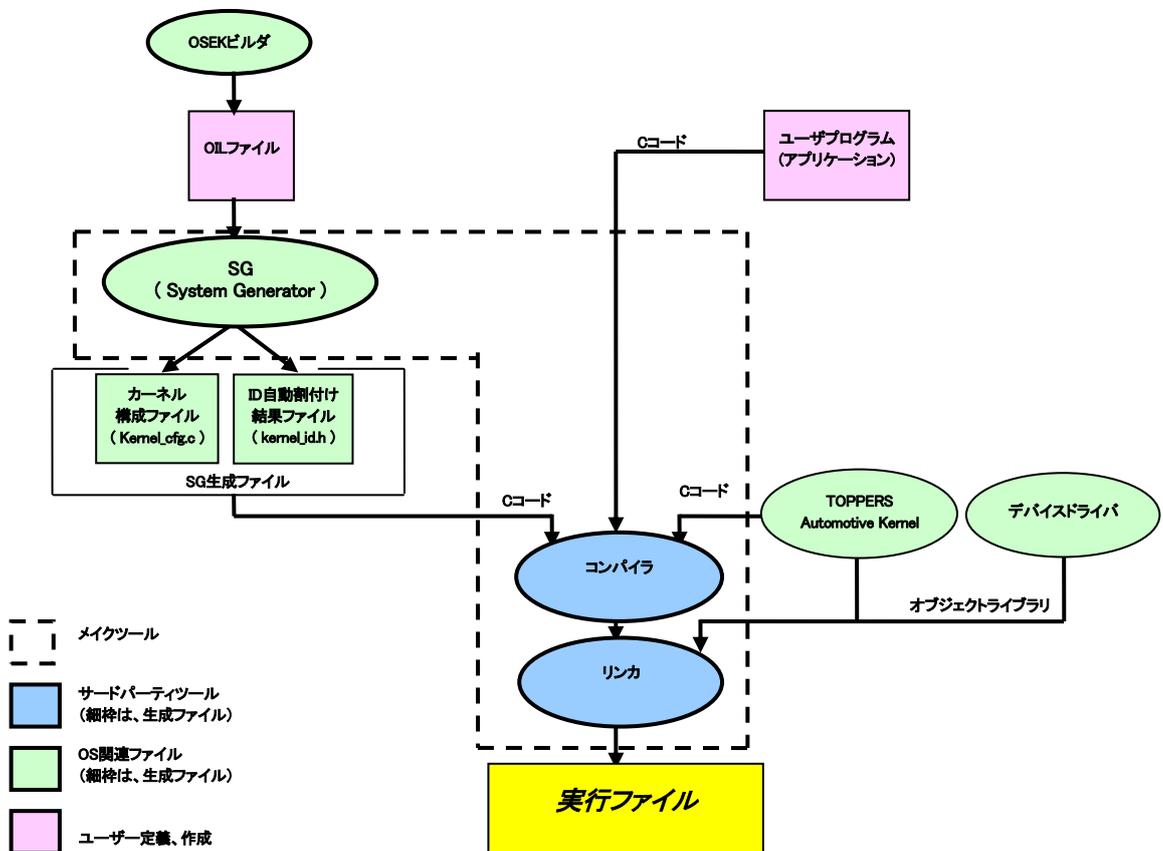
以下に、SG を用いて生成ファイルが出力される処理を示します。

※　本テキストの演習環境では、OSEK ビルダを用いません。テキストエディタで入力して、OIL ファイルを作成します。

図 5.1-2 実行ファイル生成までのファイルの流れ

SG 生成ファイル、ユーザプログラム、TOPPERS Automotive Kernel をコンパイルし、ライブラリとデバイスドライバをリンクして最終的にオブジェクトファイルを生成します。

5.2 プログラム作成手順

5.2.1 プロジェクトの新規作成

開発をはじめるためには、まず「プロジェクト」と呼ばれるプログラムを作成する領域を用意する必要があり

ます。まず、HEW をスタートメニューから起動します。HEW は統合開発環境です。正常に起動すると**図 5.2-1** の画面が表示されます。

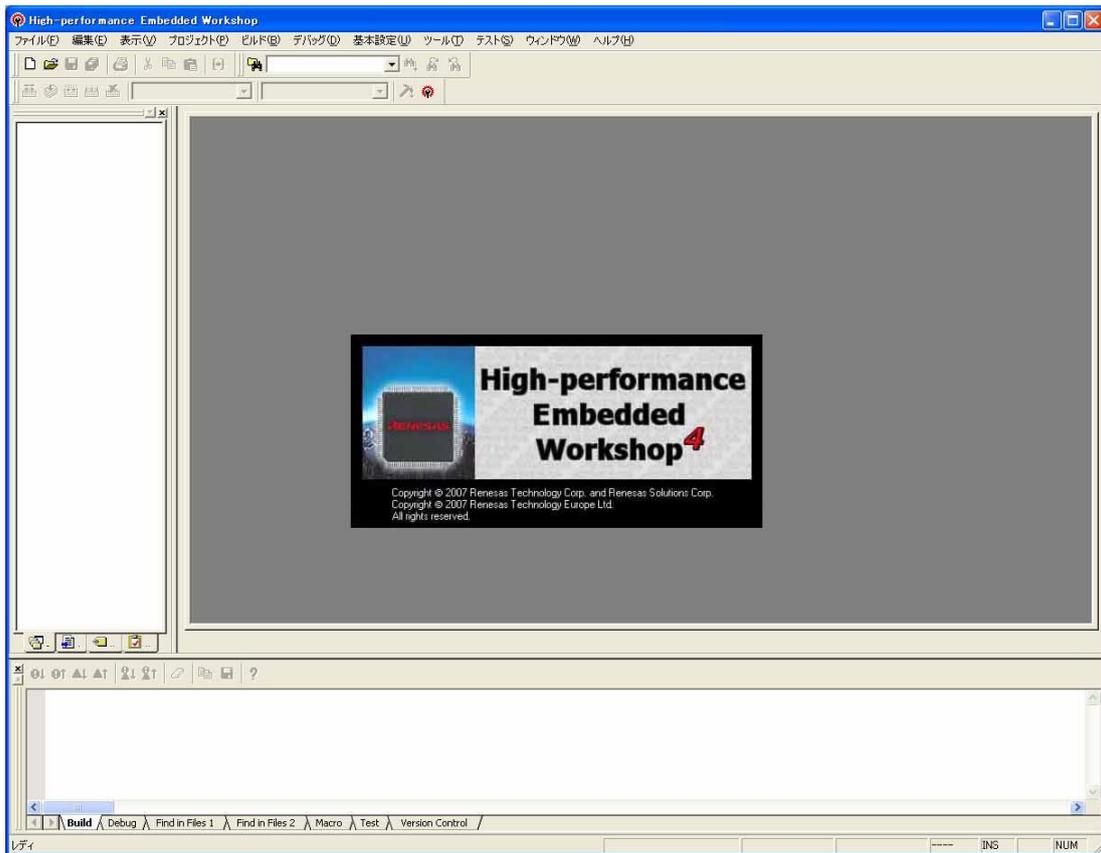



図 5.2-1 HEW 起動画面

HEW が起動したら、メニューバーから「ファイル」→「新規ワークスペース」の順に選択します（**図 5.2-2** 新規プロジェクトワークスペース画面）。

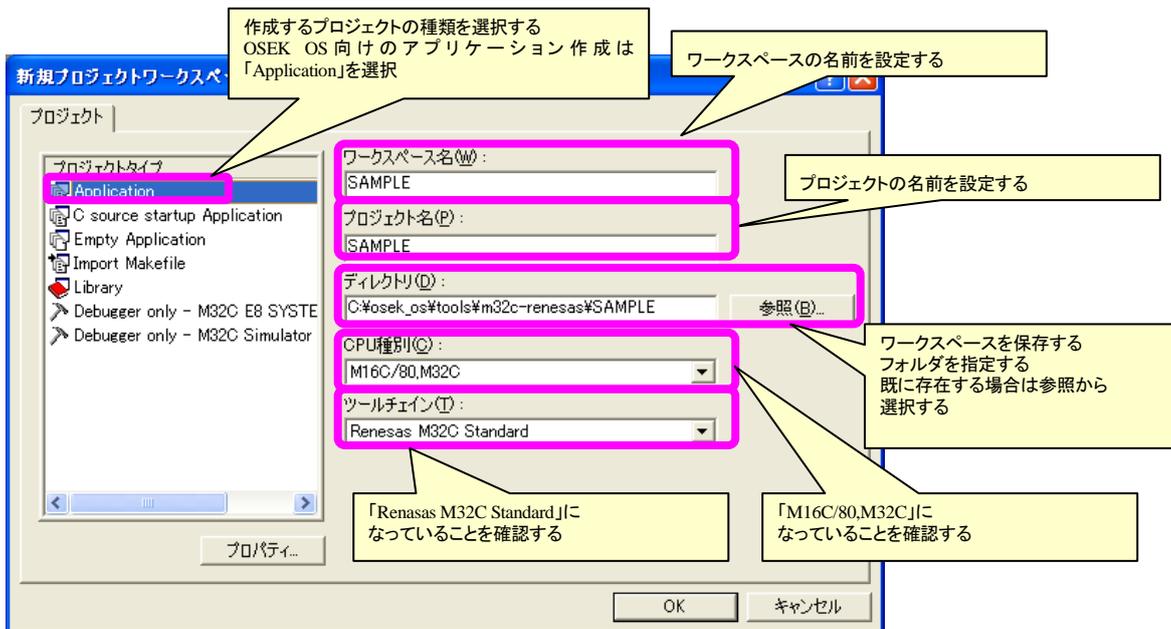



図 5.2-2 新規プロジェクトワークスペース画面

ワークスペースはプロジェクトの入れ物であり、複数のプロジェクトを管理することができます。「新規ワークスペース」を選択すると、ワークスペースを作成するための「新規プロジェクトワークスペース」ウィンドウが

表示されます。このウィンドウにて、**表 5.2-1**の様に設定していきます。

表 5.2-1 新規プロジェクトワークスペース作成の設定値

項目	設定値
プロジェクトタイプ	Application
ワークスペース名	SAMPLE(例)
プロジェクト名	SAMPLE（例）
ディレクトリ	C:¥osek_os¥tools¥m32c-renesas¥SAMPLE（例）
CPU 種別	M16C/80,M32C
ツールチェイン	Renesas M32C Standard

このとき、ワークスペース名やプロジェクト名、ディレクトリは、自由に変更可能ですが、プロジェクトタイプと CPU 種別、ツールチェインの選択は **表 5.2-1**の設定値以外を設定すると正常にビルド出来ませんので注意して下さい。また、開発環境が保存されているドライブ以外にワークスペースを用意すること（開発環境が C ドライブで、ワークスペースを D ドライブに作成する場合）も、混乱の原因になりますので避けて下さい。今回、OSEK OS を使用するのでディレクトリは tools （5.1.1章参照）の下に作成します。ワークスペースは**表 5.2-1**の設定値を設定すると、**図 5.2-2** 新規プロジェクトワークスペース画面のようになります。設定が完了したら「OK」ボタンを押します。

「New Project-1/6-Select Target CPU,Toolchain version」ウィンドウが表示されます。このウィンドウにて、**表 5.2-2**の様に設定します。

Toolchain は特定のアーキテクチャ用のビルドと開発に使用されるソフトウェアパッケージ（コンパイラやリンカなど）の集合体で、最新のものを選択します。CPU Series と CPU Group の設定値以外を設定すると正常にビルドできませんので注意して下さい。**表 5.2-2**の設定値を設定すると、**図 5.2-3** New Project-1/6-Select Target CPU,Toolchain version 画面ようになります。設定が完了したら「Next」ボタンを押します。

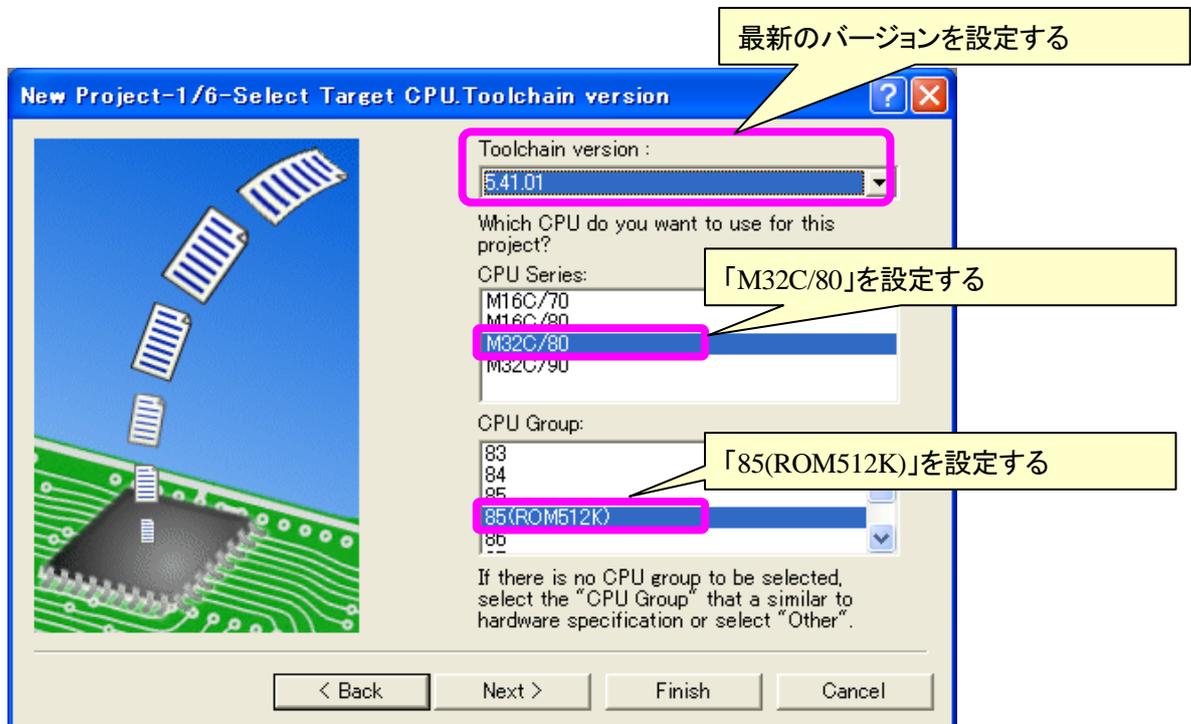



図 5.2-3 New Project-1/6-Select Target CPU,Toolchain version 画面

表 5.2-2 新規プロジェクト作成の設定値 1

項目	設定値
Toolchain version	5.41.01(最新のもの)
CPU Series	M32C/80
CPU Group	85(ROM512K)

「New Project-2/6-Select RTOS」ウィンドウが表示されます。このウィンドウにて、**表 5.2-3**の様に設定します。

OSEK OS 用のスタートアップファイルを別途追加するため、ここでは追加を行いません。**表 5.2-3**の設定値を設定すると、**図 5.2-4**のようになります。設定が完了したら「Next」ボタンを押します。

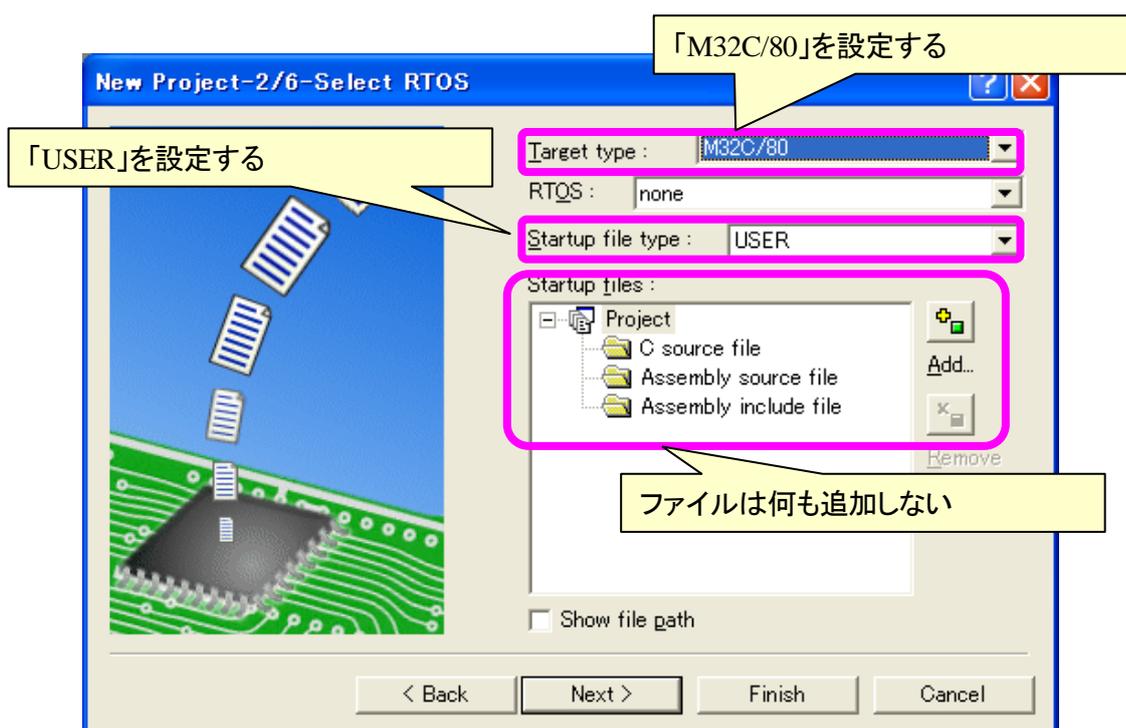



図 5.2-4 New Project-2/6-Select RTOS 画面

表 5.2-3 新規プロジェクト作成の設定値 2

項目	設定値
Target Type	M32C/80
RTOS	None
Startup file type	USER
Startup file	そのまま(追加、削除なし)
Show file path	チェックなし

「New Project-3/6-Setting the Contents of Files to be Generated」ウィンドウが表示されます。メイン関数は別途追加するため、ここではメイン関数の作成は行いません。このウィンドウにて、**表 5.2-4**の様に設定します。

表 5.2-4の設定値を設定すると、**図 5.2-5**のようになります。設定が完了したら「Next」ボタンを押します。

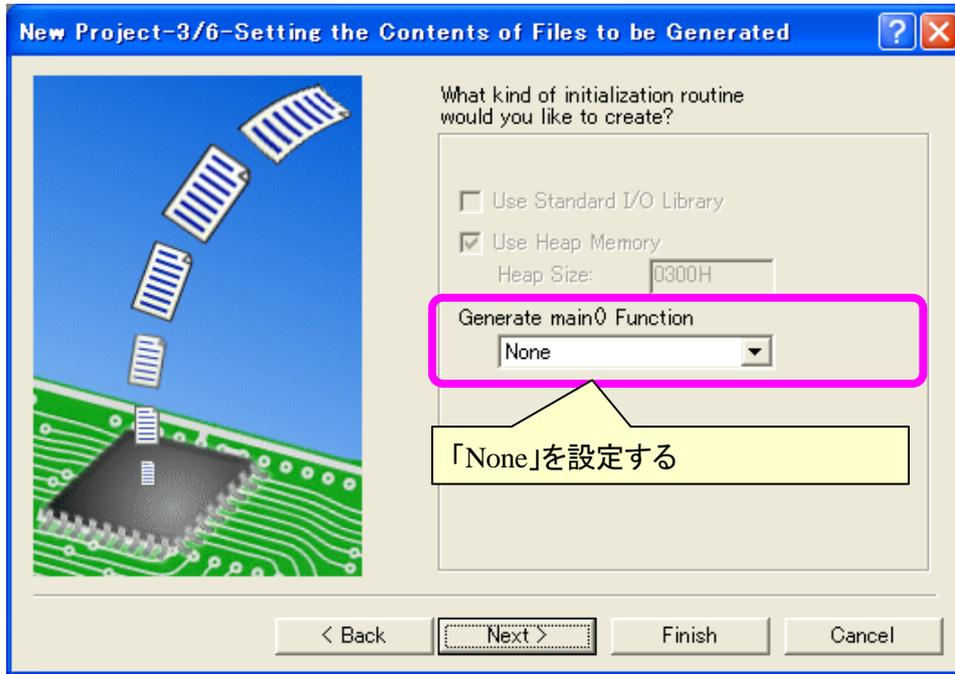



図 5.2-5 New Project-3/6-Setting the Contents of Files to be Generated 画面

表 5.2-4 新規プロジェクト作成の設定値 3

項目	設定値
Generate main() Function	None

図 5.2-6のように「New Project-4/6-Setting the Stack Area」ウィンドウが表示されます。設定項目はありません。「Next」ボタンを押します。

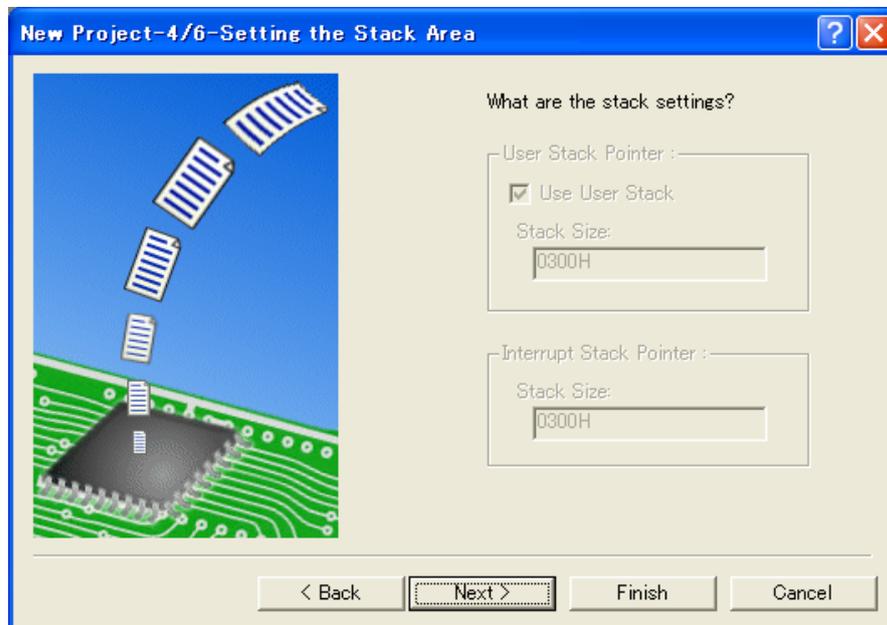



図 5.2-6 New Project-4/6-Setting the Stack Area 画面

「New Project-5/6-Setting the Target System for Debugging」ウィンドウが表示されます。書込みソフトウェア「M32C E8 SYSTEM」とシミュレータ「M32C Simulator」をプロジェクトに追加します。インストールしていない場合、ここでは表示されないので、事前にインストールしておく必要があります。尚、プロジェクト作成後に追加することも可能です。このウィンドウにて、**表 5.2-5**の様に設定します。

表 5.2-5の設定値を設定すると、**図 5.2-7**のようになります。設定が完了したら「Next」ボタンを押します。

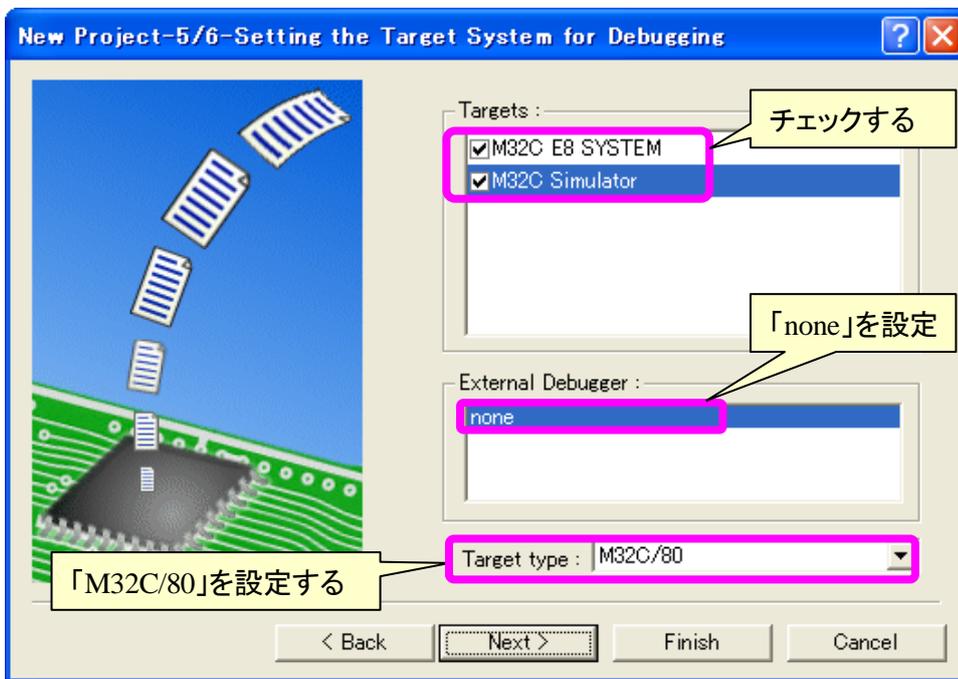



図 5.2-7 New Project-5/6-Setting the Target System for Debugging 画面

表 5.2-5 新規プロジェクト作成の設定値 4

項目		設定値
Targets	M32C E8 SYSTEM	チェックあり
	M32C Simulator	チェックあり
External Debugger		なし
Target type		M32C/80

図 5.2-8のように「New Project-6/7-Setting the Debugger Options」ウィンドウが表示されます。設定を変更する必要はありません。「Next」ボタンを押します。

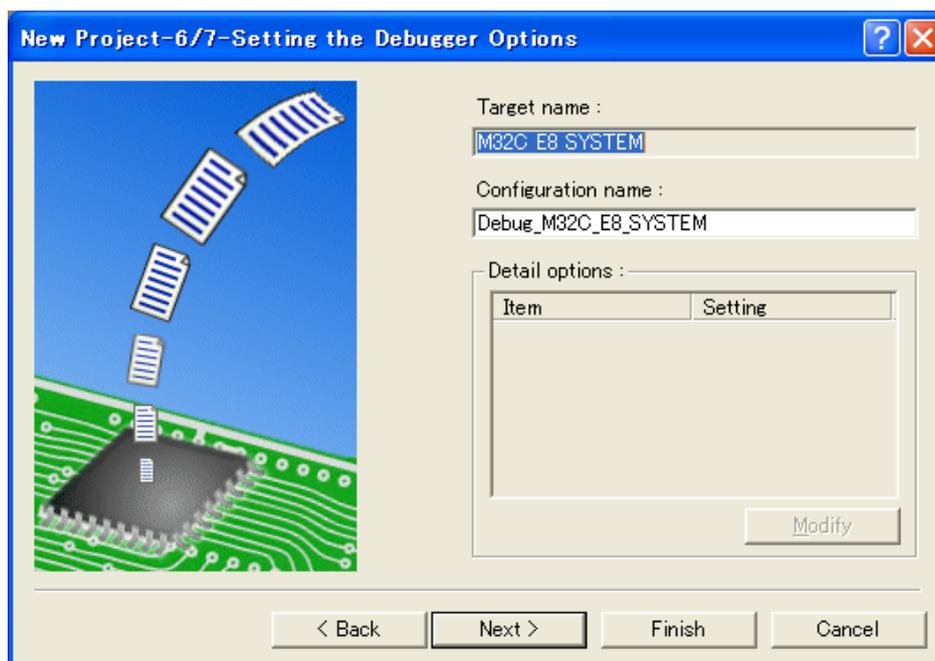



図 5.2-8 New Project-6/7-Setting the Debugger Options 画面 1

図 5.2-9のように「New Project-6/7-Setting the Debugger Options」ウィンドウが表示されます。設定を変更する必要はありません。「Next」ボタンを押します。

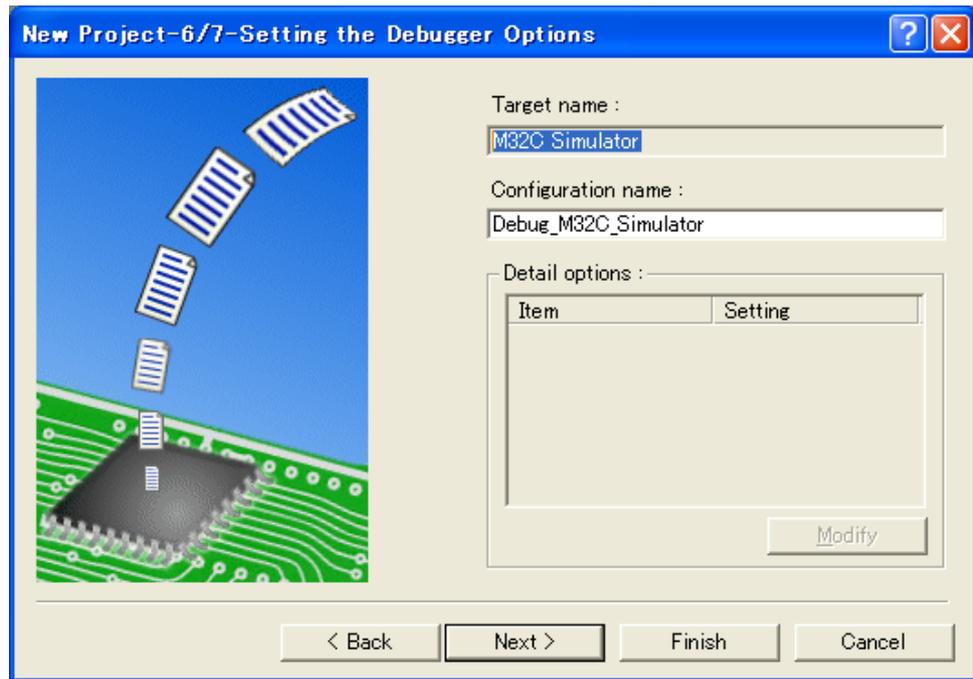



図 5.2-9 New Project-6/7-Setting the Debugger Options 画面 2

図 5.2-10のように「New Project-7/7-Changing the File Name to be Created」ウィンドウが表示されます。設定を変更する必要はありません。「Finish」ボタンを押します。

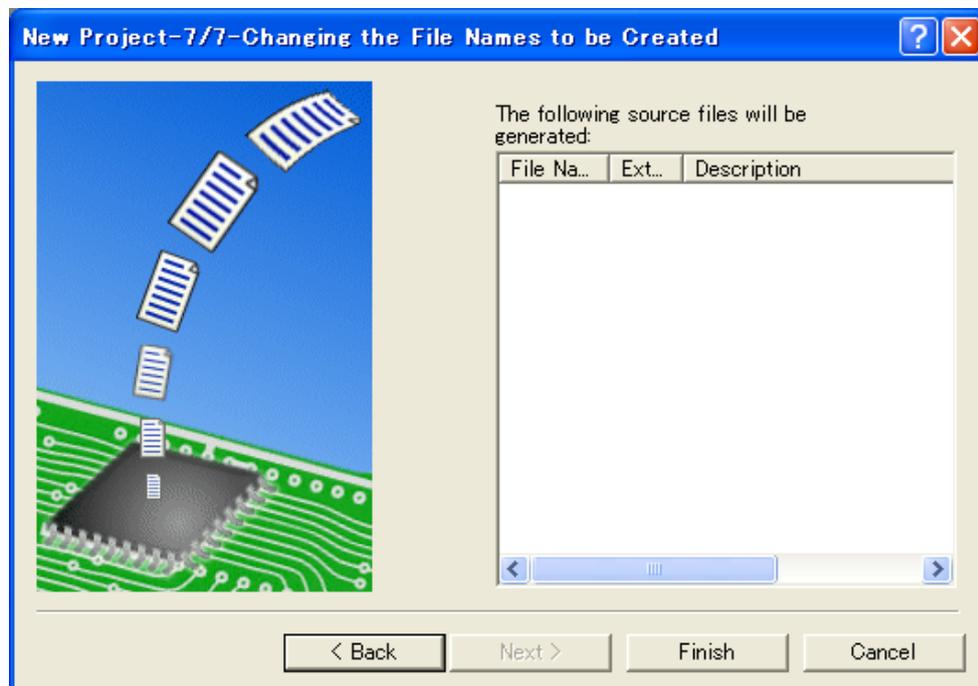



図 5.2-10 New Project-7/7-Changing the File Name to be Created 画面

図 5.2-11のように「Summary」ウィンドウが表示されます。「OK」ボタンを押します。

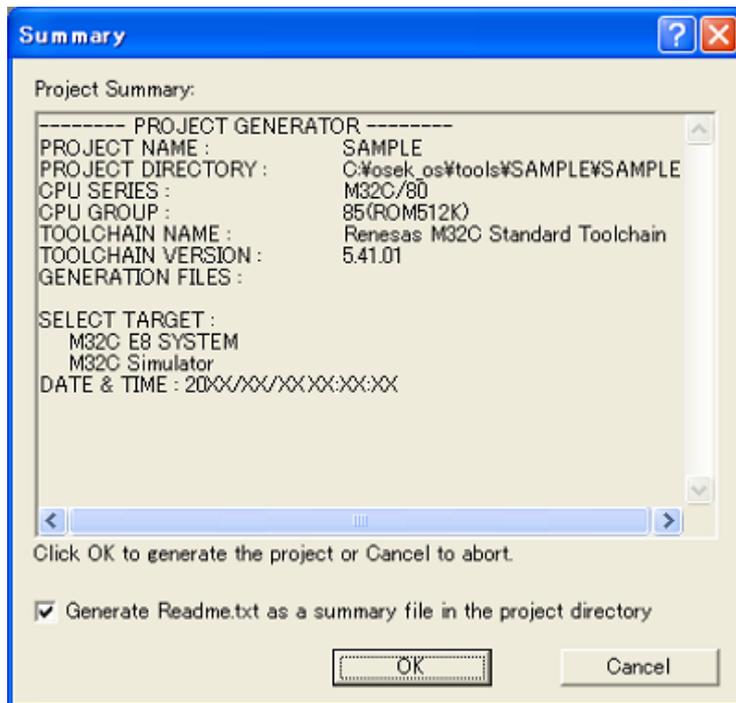



図 5.2-11 Summary 画面

5.2.2 プロジェクトの各種設定

プロジェクトの設定をするには、メニューバーから「ビルド」を選び「Runesus M32C Standard Toolchain」を選択します（**図 5.2-12** Runesus M32C Standard Toolchain 画面）。

図 5.2-12 Runesus M32C Standard Toolchain 画面

上記選択をすると「Runesus M32C Standard Toolchain」が表示されます。これから設定する内容をすべてのプロジェクト構成に対して反映するために「All Configurations」を選択します（**図 5.2-13**）。

図 5.2-13 Runesus M32C Standard Toolchain 画面

まず、「コンパイラ」タブ、「ソース」カテゴリ、「インクルードファイル検索ディレクトリ」オプション項目の「[-I]プリプロセスコマンドの# include で参照するファイルを検索するディレクトリ名を指定します」にヘッダファイルの保存ディレクトリを追加します（**図 5.2-14**）。NC308 は、ここで指定された場所に対してヘッダファイルの検索を行います。また、ここでの指定は「サブフォルダを含まない」ことに注意して下さい。つまり、あるディレクトリの下位のフォルダにも使用するヘッダファイルが存在する場合は、別途「[-I]プリプロセスコマンドの# include で参照するファイルを検索するディレクトリ名を指定します」に追加する必要があります。

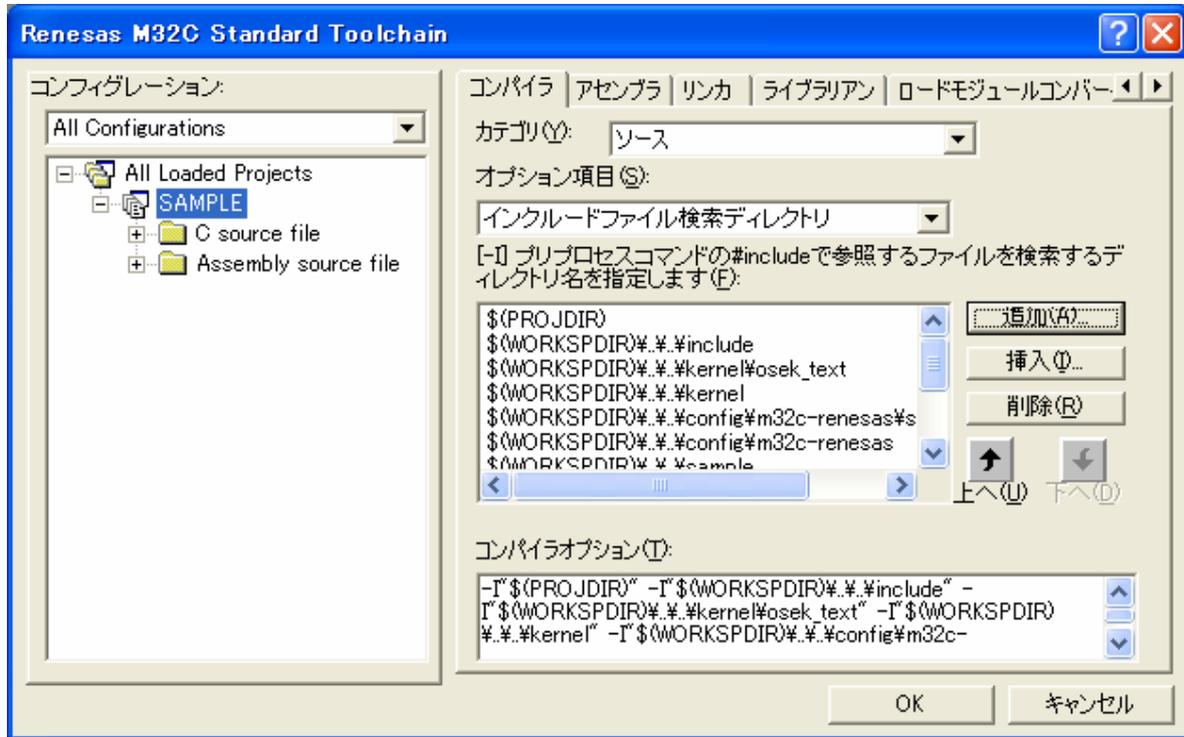



図 5.2-14 コンパイラのソース設定

今回は、**表 5.2-6**のディレクトリを追加します。追加は「追加」ボタンを押すと表示される「Add include file directory」ウィンドウにて行います（**図 5.2-15**）。

図 5.2-15 Add include file directory 画面

表 5.2-6 追加するインクルードファイル検索ディレクトリ

相対パス	サブディレクトリ
Project directory	-(なし)
WorkSpace directory	..¥..¥include
WorkSpace directory	..¥..¥kernel¥ecc2
WorkSpace directory	..¥..¥kernel
WorkSpace directory	..¥..¥config¥m32c-renesas¥s810-tpf-85
WorkSpace directory	..¥..¥config¥m32c-renesas
WorkSpace directory	..¥..¥sample
WorkSpace directory	..¥..¥syslib¥m32c-renesas¥s810-tpf-85
WorkSpace directory	..¥..¥syslib
WorkSpace directory	..¥..¥driver¥led
WorkSpace directory	..¥..¥driver¥led¥s810-tpf-85
WorkSpace directory	..¥..¥driver¥sw
WorkSpace directory	..¥..¥driver¥sw¥s810-tpf-85

次に、「コンパイラ」タブ、「オブジェクト」カテゴリの「デバッグオプション」について設定します。「[-g]デバッグ情報をアセンブラ言語ソースファイル(拡張子「.a30」)に出力します。」にチェックを入れます(**図 5.2-16**)。デバッグ情報はシミュレータでデバッグをする際に必要となる情報です。そのためリリース版では不要な情報なので、コンフィギュレーションの「Release」についてはチェックを外します。

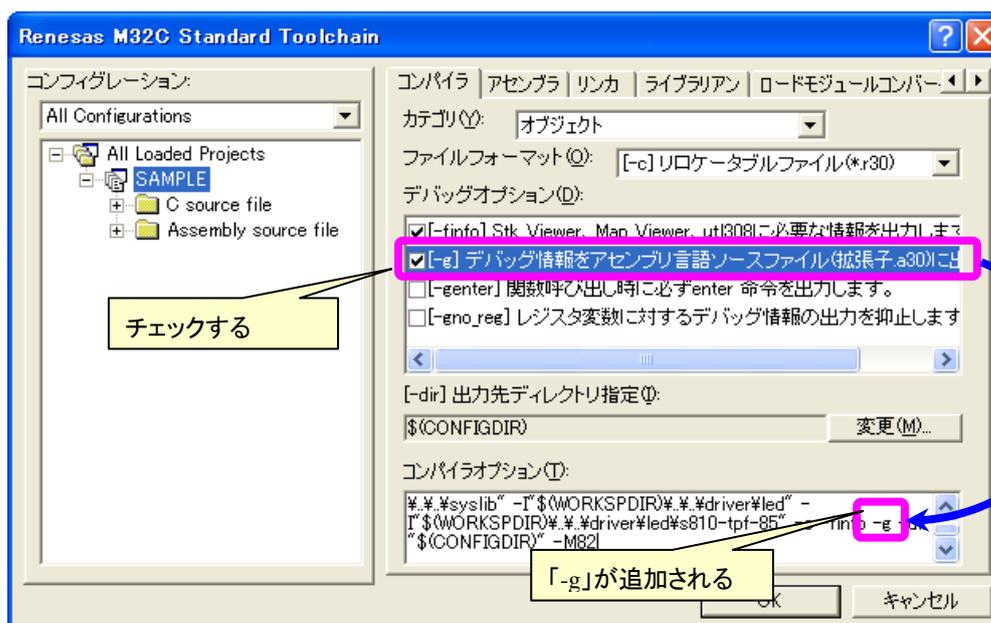



図 5.2-16 コンパイラのオブジェクト設定

次に、「コンパイラ」タブ、「リスト」カテゴリの設定をします。「[-dS]C 言語ソースリストをコメントして出力したアセンブリ言語ソースファイル(拡張子「.a30」)を生成します(アセンブル後も削除しません)。」にチェックを入れます(**図 5.2-17**)。シミュレータでデバッグをする際にアセンブリ言語ソースファイルを必要とするため、アセンブル後も削除しないようにします。

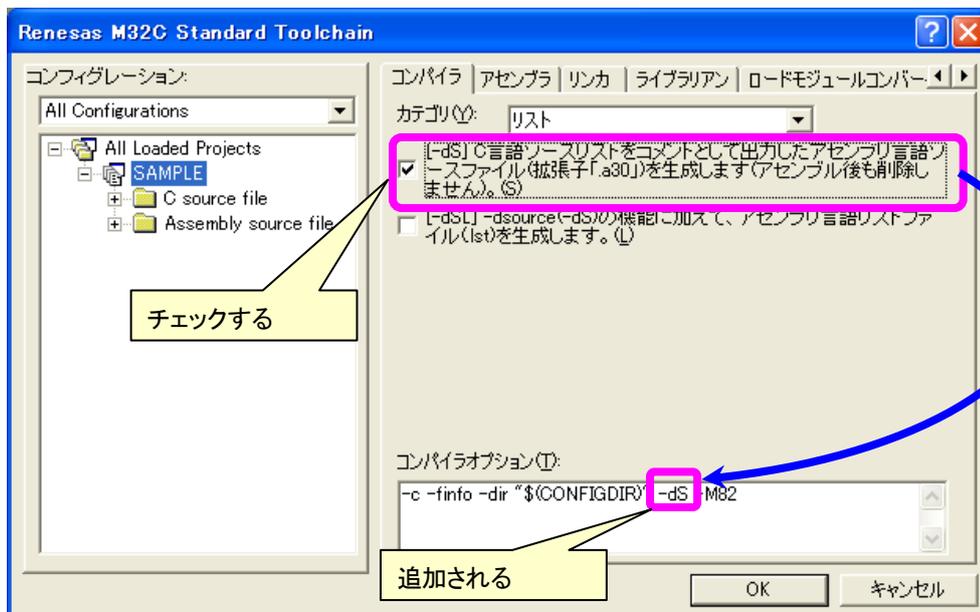



図 5.2-17 コンパイラのリスト設定

次に、「コンパイラ」タブ、「最適化」カテゴリに変更し、最適化オプションについて設定します。

最適化は、ユーザが記述したプログラムを「実行スピード向上」または「オブジェクトサイズの減少」について考慮し、最適なコードをアセンブリ命令として出力する機能です。

最適化の選択は「サイズとスピード」と「最適化レベル」で行います。今回は、デバッグ情報に影響を与えないレベルの最適化をするようにするために「最適化レベル」にチェックを入れ、「[-O2]-O と同じです。」を選択します（**図 5.2-18**）。

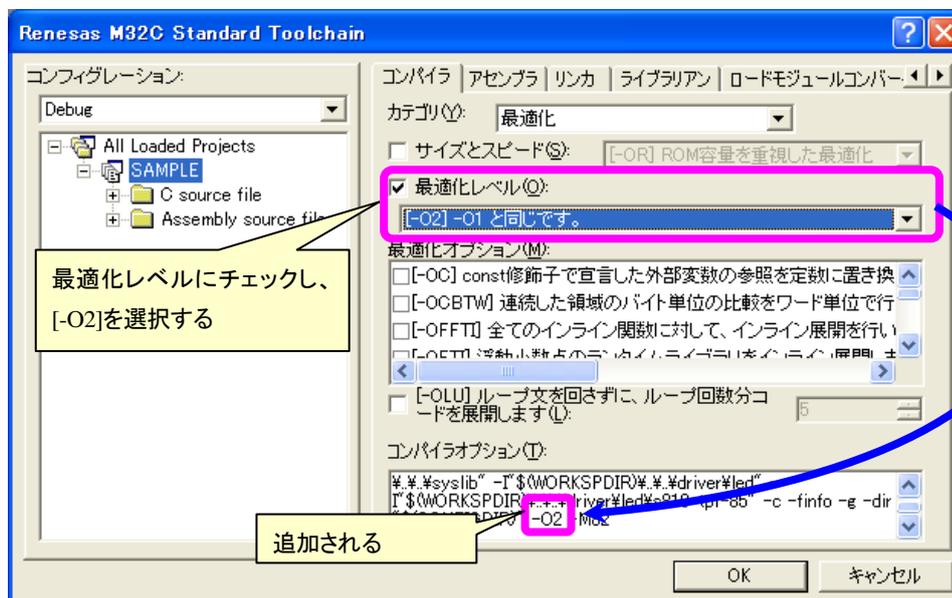



図 5.2-18 コンパイラの最適化設定

次に、「コンパイラ」タブ、「警告」カテゴリに変更し、警告オプションについて設定します。

警告は、コンパイル時に言語仕様に関する記述の間違いに対して警告(ワーニングメッセージ)を出力する機能です。今回は、問題のある記述に対してすべて警告するようにするために、「警告オプション」の「[-Wall]検出可能な警告（"-Wlarge_to_small"、"Wno_used_argument"で出力される警告を除く）をすべて表示します。」にチェックを入れます。

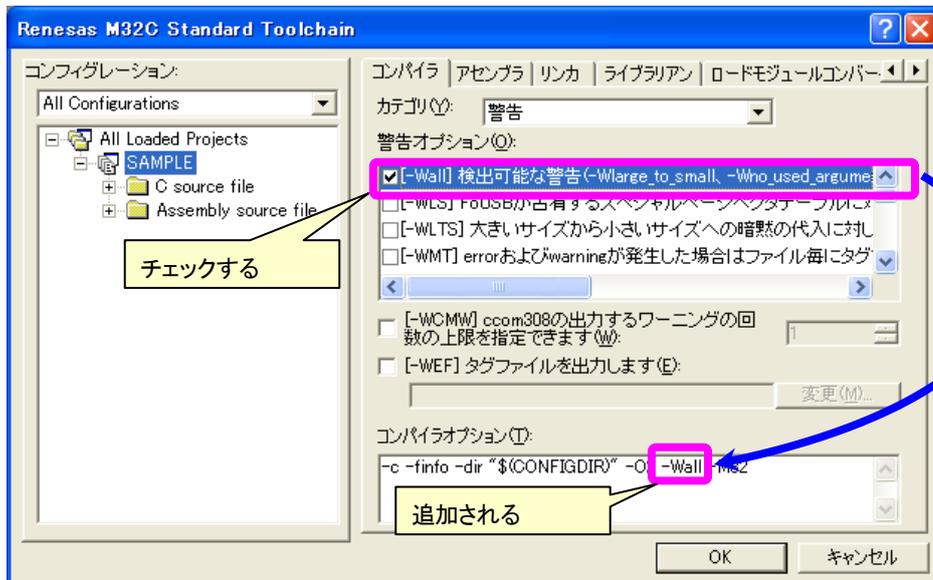



図 5.2-19 コンパイラの警告設定

次に、「アセンブラ」タブ、「ソース」カテゴリ、「インクルードファイル検索ディレクトリ」オプション項目の「[-I]インクルードファイルの検索ディレクトリを指定する。」にアセンブラが参照するヘッダファイルの保存ディレクトリを設定します（**図 5.2-20**）。AS308 は、ここで指定された場所に対してヘッダファイルの検索を行います。今回は、**表 5.2-7**のディレクトリを設定します。設定は「変更」ボタンを押すと表示される「Include file directory」ウィンドウにて行います（**図 5.2-21**）。

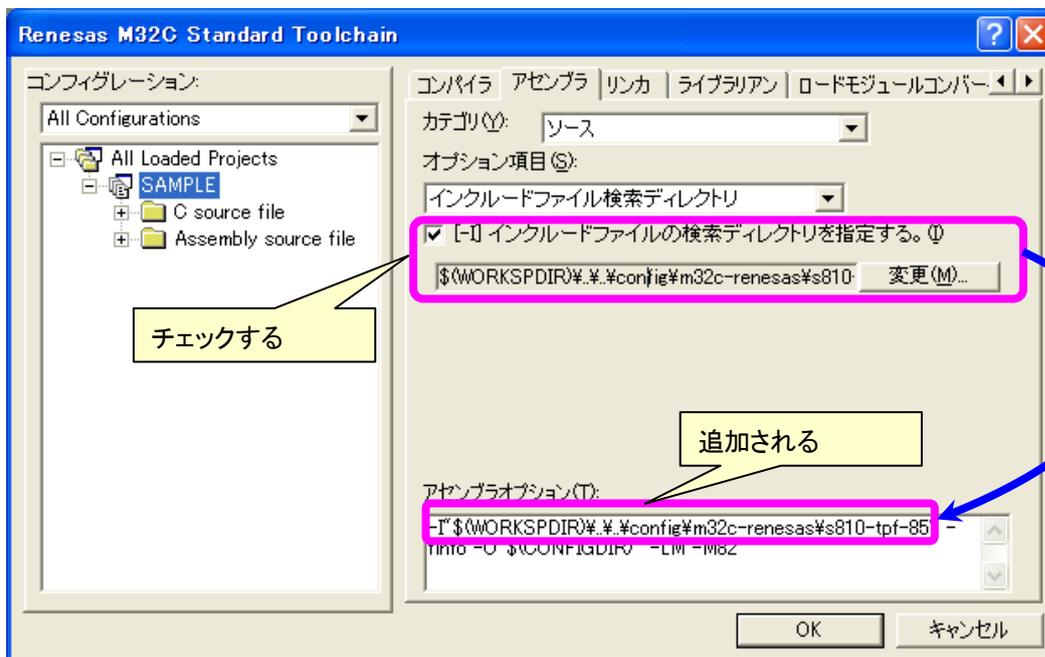



図 5.2-20 アセンブラのソース設定

表 5.2-7 設定するインクルードファイル検索ディレクトリ

相対パス	サブディレクトリ
WorkSpace directory	..¥..¥config¥m32c-renesas¥s810-tpf-85

図 5.2-21 Include file directory 画面

次に、「リンカ」タブ、「出力」カテゴリの「マップファイルを生成する」について設定します。マップファイルは作成したプログラムのメモリや関数のアドレスなどのリンク情報が記されたテキストファイルです。「[-MSL]16 文字を超えるシンボルをそのままマップファイルに出力」を選択します。以上で、すべての設定が完了しました。最後に「OK」ボタンを押して、設定内容を確定します。

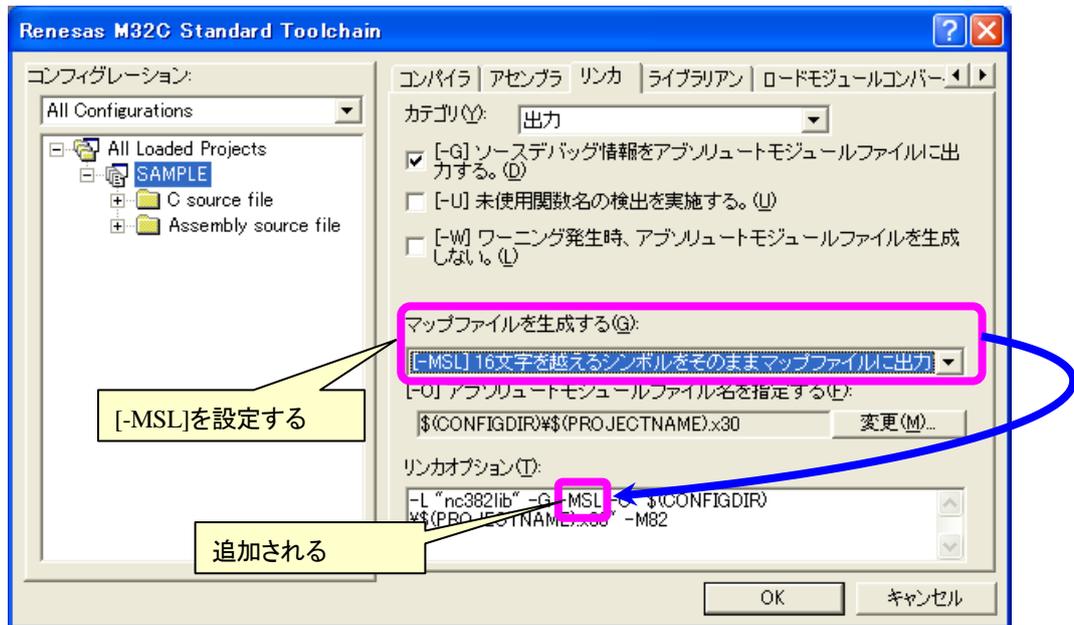



図 5.2-22 リンカの出力設定

5.2.3 プロジェクトへのファイル登録

TOPPERS Automotive Kernel をプロジェクトに追加します。まず、プロジェクトに追加するファイルを見やすくするために、フォルダを作成します。ツリーから「SAMPLE」プロジェクトを右クリックし、「フォルダの追加」を選択します（**図 5.2-23**）。

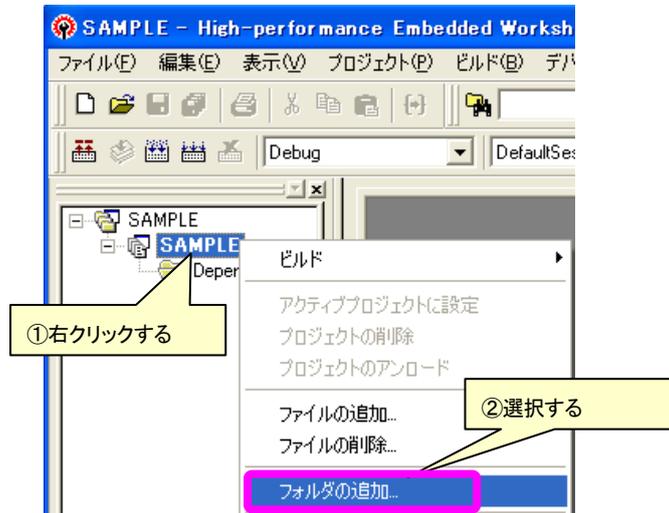



図 5.2-23 フォルダの追加選択

「フォルダの追加」ウィンドウが表示されます。フォルダ名に「osek_os」を設定し、「OK」ボタンを押します（**図 5.2-24**）。

図 5.2-24 フォルダの追加画面

「osek_os」フォルダが「SAMPLE」プロジェクトの下に追加されます（**図 5.2-25**）。

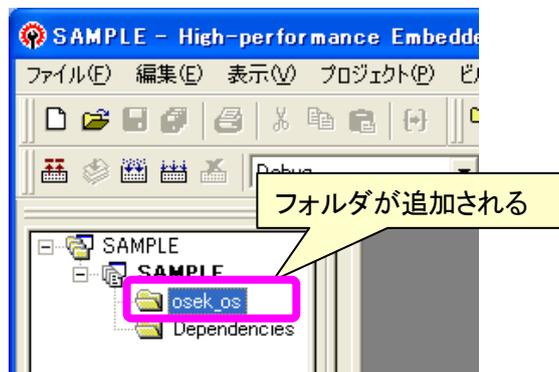



図 5.2-25 OSEK_OS フォルダの追加

上記で追加したフォルダの下にファイルを追加します。ファイルを追加する場合は、ツリーから「osek_os」フォルダを右クリックし、「ファイルの追加」を選択します（**図 5.2-26**）。

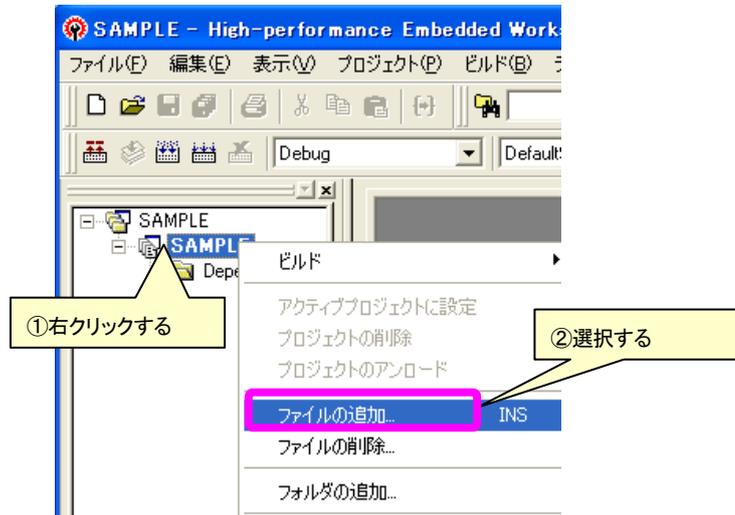



図 5.2-26 ファイルの追加選択

ツリーの「osek_os」の中に「config」フォルダをドラッグ&ドロップします（図 5.2-27）。

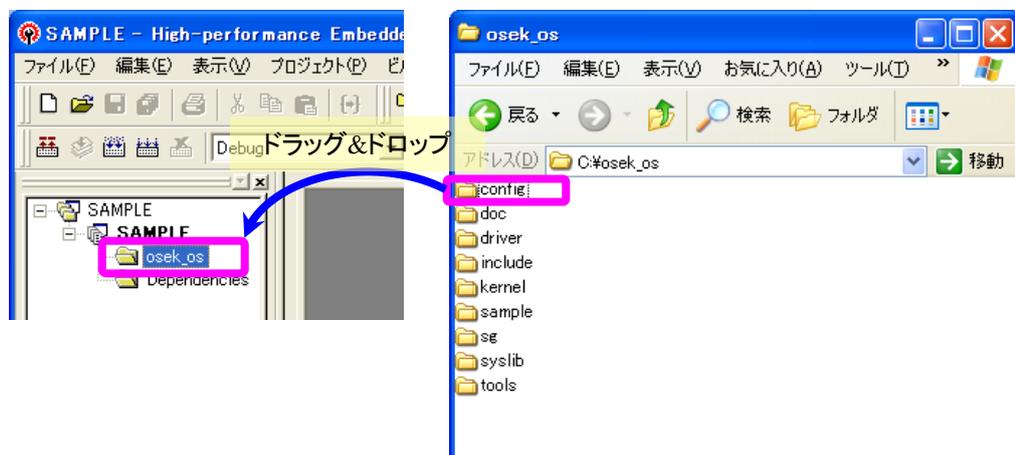



図 5.2-27 config フォルダのドラッグ&ドロップ

ドラッグ&ドロップすると「フォルダからファイルの追加」ウィンドウが表示されます。

「config」フォルダ以下のすべてのソースファイルを追加するため、すべての項目にチェックをし、「OK」ボタンを押します（図 5.2-28）。

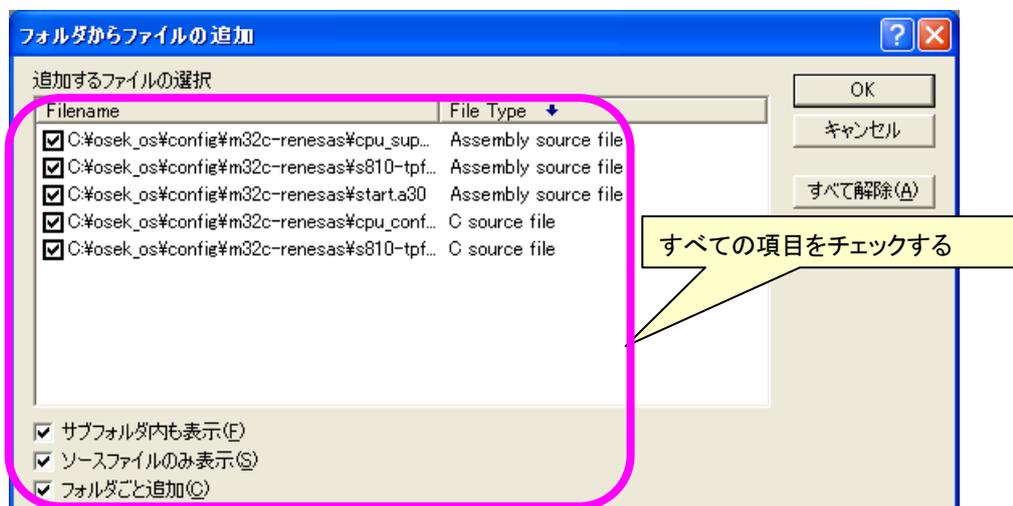



図 5.2-28 フォルダからファイルの追加画面

ツリーに「config」フォルダ以下のファイルが追加されます（**図 5.2-29**）。

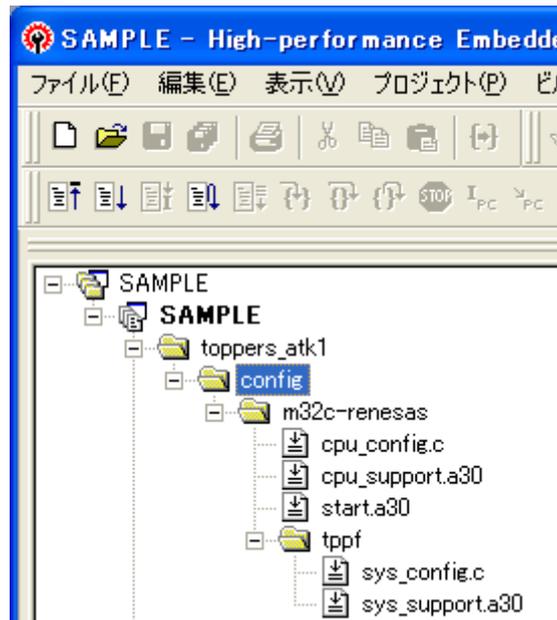



図 5.2-29 config フォルダの追加

同様の方法で「osek_os」の下に driver、kernel、sample、syslib、tools を追加します。

5.2.4 システムジェネレータの使用方法

SG の実行は、コマンドプロンプト上で行われます。実行時にはコマンドラインで、SG 実行ファイルと OIL ファイル、コマンドラインオプションの指定を行う必要があります。実行する度にコマンドラインを指定することも可能ですが、バッチファイルを作成し、事前に必要なコマンドを設定しておくと便利です。今回は、プロジェクト（「SAMPLE」フォルダ）の下に「sample_sg.bat」という名前のバッチファイルを作成します。

バッチファイルには次のように記述します。また今回、使用するコマンドオプションの内容は**表 5.2-8**に示します。

```
@del .¥kernel_cfg.c
@del .¥kernel_id.h

..¥..¥..¥sg¥sg.exe sample.oil ¥
 -template=..¥..¥..¥config¥m32c-renesas¥s810-tpf-85¥m32c85.sgt ¥
 -I .¥..¥..¥..¥sg¥impl_oil -I .¥..¥..¥..¥syslib -I .¥..¥..¥..¥syslib¥m32c-renesas¥s810-clg3-85 -os=ECC2
```


表 5.2-8 コマンドラインオプション

オプション	内容
-template	ターゲット依存の情報が記載されたテンプレートファイルを指定
-I	OIL ファイルがインクルードするファイルが存在するディレクトリのパスを指定
-os	コンフォーマンスクラスを指定(BCC1/ BCC2/ ECC1/ ECC2)

「@del .¥kernel_cfg.c」と「@del .¥kernel_id.h」で、既に SG の出力ファイルがある場合に削除します。ファイルがない場合は、何もしません。

「sg.exe」が SG の実行ファイル名です。実行ファイル名は最初に書きます。「sample.oil」は**5.2.6 OIL ファイルの作成**の項目で作成した OIL ファイルです。SG の実行ファイルの次に書きます。コマンドラインオプション

は OIL ファイルの後に記載します。

5.2.5 プロジェクトへのシステムジェネレータ登録

ビルド時に SG の処理を自動実行できるようにするために、プロジェクトに SG を登録します。ます、メニューバーから「ビルド」を選び「ビルドフェーズ」を選択します（**図 5.2-30**）。

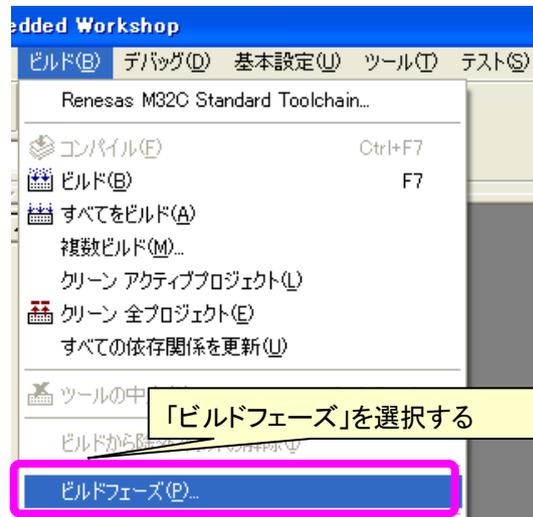



図 5.2-30 ビルドフェーズの選択

上記選択をすると「ビルドフェーズ」ウィンドウが表示されます。ビルドフェーズに SG を追加します。「ビルド順序」タブの「追加」ボタンを押します（**図 5.2-31**）。

図 5.2-31 ビルドフェーズ順序の追加

「ビルド順序」タブの「追加」ボタンを押すと、「新規ビルドフェーズ-1/4 ステップ」ウィンドウが表示されます。今回は、新規でビルドフェーズを作成するので「新規カスタムフェーズの作成」を設定します。設定が完了したら「次へ」ボタンを押します（**図 5.2-32**）。

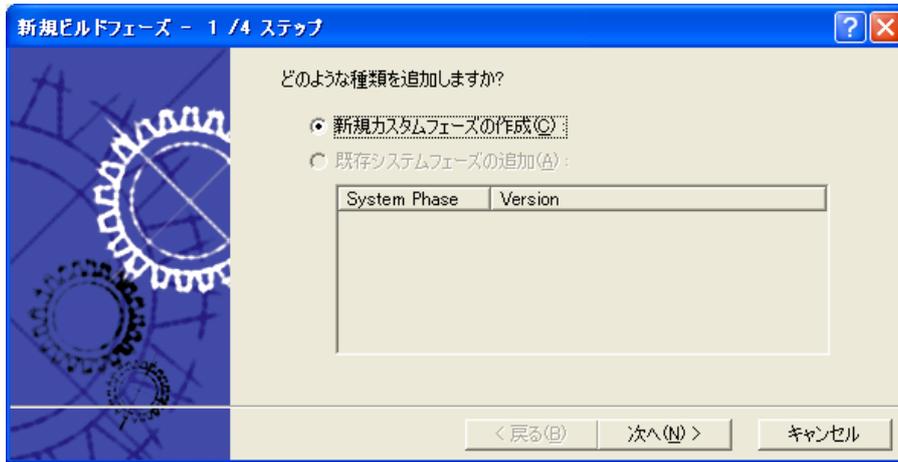



図 5.2-32 新規ビルドフェーズ-1/4 ステップ画面

「新規ビルドフェーズ-2/4 ステップ」ウィンドウが表示されます。SG はビルド時に 1 度だけ実行すれば良いので、「単一フェーズ」を選択します。設定が完了したら「次へ」ボタンを押します（**図 5.2-33**）。

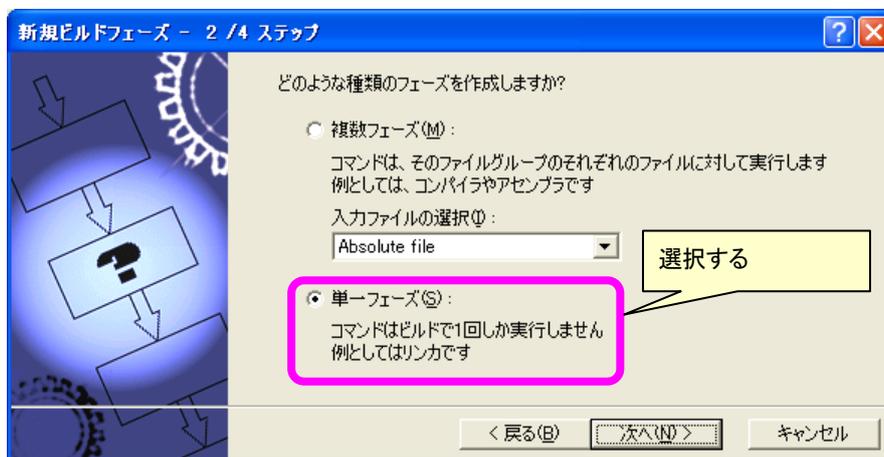



図 5.2-33 新規ビルドフェーズ-2/4 ステップ画面

「新規ビルドフェーズ-3/4 ステップ」ウィンドウが表示されます。このウィンドウに対して、**表 5.2-9**の様に設定します。

表 5.2-9 新規ビルドフェーズ作成の設定値

項目	設定値
フェーズ名	SystemGenerator (例)
コマンド	$(PROJDIR)¥sample_sg.bat
デフォルトオプション	-(設定しない)
初期ディレクトリ	$(PROJDIR)

コマンドには**5.2.4 システムジェネレータの使用方法** で作成した SG 実行用のバッチファイル「sample_sg.bat」を指定します。今回は、バッチファイルをプロジェクトの下に作成したので、コマンド、出デフォルトオプション、初期ディレクトリの設定は**表 5.2-9**のように設定します（**図 5.2-34**）。

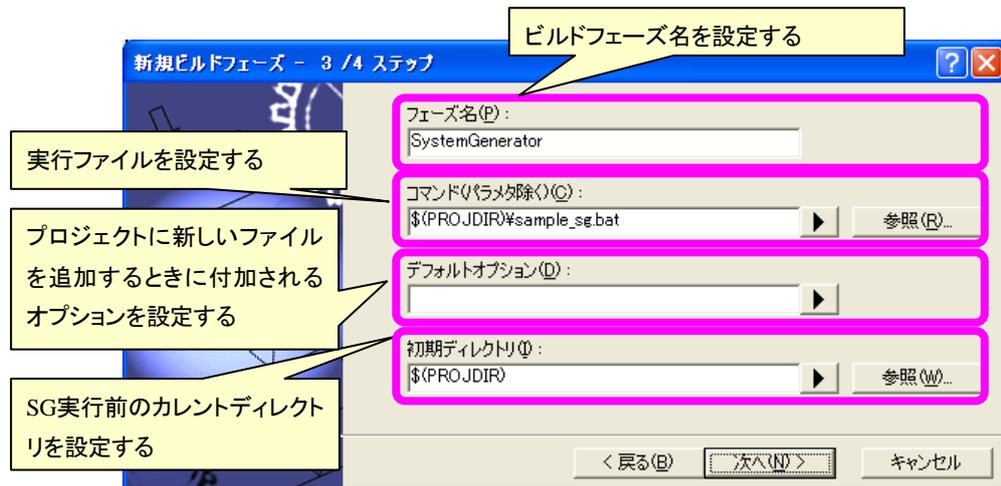



図 5.2-34 新規ビルドフェーズ-3/4 ステップ画面

「新規ビルドフェーズ-4/4 ステップ」ウィンドウが表示されます。ここではコマンド実行時に必要な環境変数の設定が行えます。今回は、設定せず、「完了」ボタンを押します（**図 5.2-35**）。

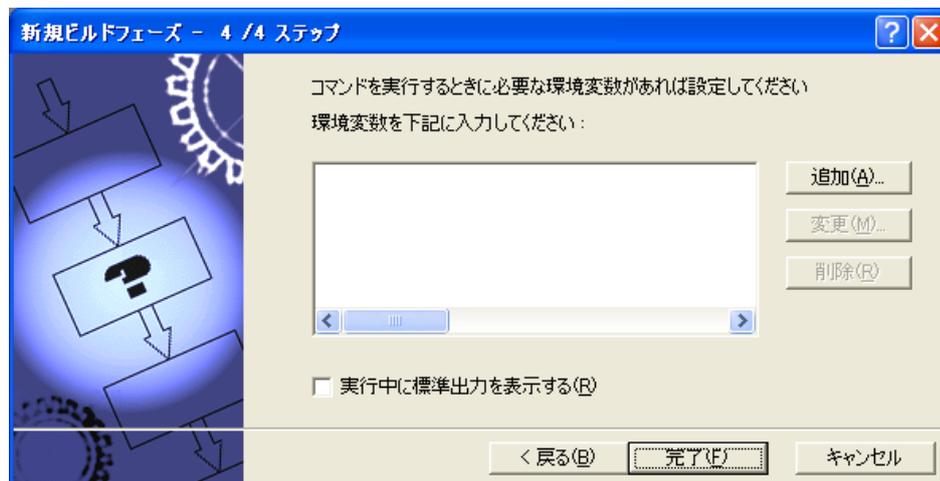



図 5.2-35 新規ビルドフェーズ-4/4 ステップ画面

SystemGenerator が「ビルドフェーズの順序」に追加されます。SG が生成したファイルもコンパイル対象となるため、コンパイラより先に SG を実行する必要があります。SystemGenerator を選択した状態で「上へ」ボタンを押して SystemGenerator を一番上まで移動します。最後に「OK」ボタンを押して、設定内容を確定します（**図 5.2-36**）。

図 5.2-36 ビルドフェーズ順序の変更

5.2.6 OIL ファイルの作成

OIL ファイルは 1 つのメイン OIL ファイルとそこからインクルードされるファイルで構成されます。今回は、メイン OIL ファイルをプロジェクトフォルダ（SAMPLE）の下に、「sample.oil」という名前で作成します。

OIL の記述は大きく分けて 3 部構成となっています。はじめに、準拠している OIL 仕様のバージョン情報を定義（OIL バージョン部）、次に標準的な実装仕様を示す定義（実装部）、最後に特定の CPU に配置されたアプリケーションの構造定義（アプリケーション部）です。OIL 記述では、# include、C++と同様のコメントが使用可能です。

OIL バージョン部と実装部は SG を実行した際に、「implementation.oil」ファイルとして自動生成されます。そのため、メイン OIL ファイルの最初で「implementation.oil」をインクルードします。

```
#include "implementation.oil"
```

アプリケーション部では、ユーザアプリケーションの実装に合わせたオブジェクトの状態を記述します。今回は、1 つのタスクを起動するアプリケーションで必要な記述について記載します。

CPU オブジェクトを作成します。CPU オブジェクトはすべてのオブジェクトのコンテナとして使用します。他のオブジェクトは CPU オブジェクトの中に作成していくことになります。オブジェクト名は任意ですが、今回は「current」とします。

```
CPU current {

}
```

次に、サンプルプログラム用ライブラリを syslib フォルダに用意してあるので、ライブラリ用の OIL ファイルをインクルードします。ライブラリはシリアル機能、100μs タイマ機能の 2 種類があります。

```
CPU current {
        #include <serial.oil>
        #include <t100us_timer.oil>
}
```

次に、OS オブジェクトを記述します。OS オブジェクトは OS プロパティを定義するのに使用されるオブジェ

クトで、CPU オブジェクト内に 1 つ定義しなければなりません。OS オブジェクトの属性を**表 5.2-10**の様に設定していきます。

表 5.2-10 OS オブジェクトの設定

属性	設定値
STATUS	STANDARD
STARTUPHOOK	FALSE
ERRORHOOK	FALSE
SHUTDOWNHOOK	FALSE
PRETASKHOOK	FALSE
POSTTASKHOOK	FALSE
USEGETSERVICEID	TRUE
USEPARAMETERACCESS	TRUE
USERESSCHEDULER	FALSE

STATUS 属性では、システムサービスの戻り値で取得エラーの種類を選択します。エラーの種類は標準エラー（STANDARD）と拡張エラー（EXTENDED）があり、デバッグ時など動的なチェックが不可欠な場合は拡張エラー、リリース時など最低限のエラーチェックを行う場合は標準エラーを選択するといった使い方をします。STARTUPHOOK~ POSTTASKHOOK 属性は各種フックルーチンの使用有無を設定します。今回は、フックルーチンを使用しないので FALSE を設定します。

USEGETSERVICEID 属性と USEPARAMETERACCESS 属性はエラー情報取得マクロの使用有無を設定します。ただし、現状の TOPPERS Automotive Kernel では FALSE は未サポートとなっているため、必ず TRUE を設定する必要があります。USERESSCHEDULER 属性はスケジューラリソース（ディスパッチの禁止）の使用有無を設定します。またオブジェクト名は任意ですが、今回は「os」とします。

```
CPU current {
                    : (省略)
        OS os {
                STATUS = STANDARD;
                STARTUPHOOK = FALSE;
                ERRORHOOK = FALSE;
                SHUTDOWNHOOK = FALSE;
                PRETASKHOOK = FALSE;
                POSTTASKHOOK = FALSE;
                USEGETSERVICEID = TRUE;
                USEPARAMETERACCESS = TRUE;
                USERESSCHEDULER = FALSE;
        };
}
```

次に、APPMODE オブジェクトを記述します。APPMODE オブジェクトはアプリケーションの異なるモードの操作を定義するのに使用されるオブジェクトで、CPU オブジェクト内に 1 つ以上定義しなければならないことになっています。本オブジェクトには属性は存在しません。オブジェクト名は任意ですが、今回は「AppMode1」とします。

```
CPU current {
                    : (省略)
        APPMODE AppMode1 {};
}
```

最後に、TASK オブジェクトを記述します。タスク情報を定義するために使用されるオブジェクトです TASK オブジェクトの属性を**表 5.2-11**の様に設定します。

表 5.2-11 TASK オブジェクトの設定

属性	設定値
AUTOSTART	TRUE
APPMODE	AppMode1
PRIORITY	14
STACKSIZE	0x0180
ACTIVATION	1
SCHEDULE	NON

AUTOSTART 属性では、システム初期化時にタスクを自動起動するか否かを設定します。自動起動する場合、APPMODE 属性にて自動実行するアプリケーションモードを設定します。今回は、AppMode1 で自動起動するタスクにするので AUTOSTART 属性を TRUE 、APPMODE 属性を AppMode1 にします。

PRIORITY 属性では、タスクの優先度を設定します。優先度は 0〜15 の範囲で設定可能で、値が大きいほど優先度が高くなります。

STACKSIZE 属性では、タスクで使用するスタックサイズを設定します。

ACTIVATION 属性では、多重起動要求の最大数を設定します。今回は、多重起動しないので 1 を設定します。

SCHEDULE 属性では、スケジューリング方法（フルプリエンプティブスケジュール（FULL)、ノンプリエンプティブスケジュール（NON)）を設定します。またオブジェクト名は任意ですが、今回は「MainTask」とします。

```
CPU current {
                    : (省略)
        TASK MainTask {
                AUTOSTART = TRUE {
                        APPMODE = AppMode1;
                };
                PRIORITY = 14;
                STACKSIZE = 0x0180;
                ACTIVATION = 1;
                SCHEDULE = NON;
        };
}
```

最終的に OIL ファイルは次のようになります。

```
#include "implementation.oil"

CPU current {
	#include <serial.oil>
	#include <t100us_timer.oil>

	OS os {
		STATUS = STANDARD;
		STARTUPHOOK = FALSE;
		ERRORHOOK = FALSE;
		SHUTDOWNHOOK = FALSE;
		PRETASKHOOK = FALSE;
		POSTTASKHOOK = FALSE;
		USEGETSERVICEID = TRUE;
		USEPARAMETERACCESS = TRUE;
		USERESSCHEDULER = FALSE;
	};

	APPMODE AppMode1 {};

	TASK MainTask {
		AUTOSTART = TRUE {
			APPMODE = AppMode1;
		};
		PRIORITY = 14;
		STACKSIZE = 0x0180;
		ACTIVATION = 1;
		SCHEDULE = NON;
	};
};
```

5.2.7 アプリケーションの作成

本来、アプリケーションは新規にファイルを作成しそこに記述しますが、今回は sample フォルダ内の sample.h 及び sample.c 内に記述します。

OIL ファイルで定義したタスク「MainTask」を使用するアプリケーションを作成します。まず、sample.h の記述を行います。SG が自動生成した ID 自動割付け結果ファイル「kernel_id.h」を定義します。このファイルは、他のタスクを制御する場合などに、生成された ID で操作するために必要なファイルです。

```
#include "kernel_id.h"
```

OIL 定義シンボルの外部参照を定義します。タスク識別子をプログラム内で使用する場合に必要です。

```
DeclareTask( MainTask );
```

次に、sample.c の記述を行います。タスクを記述するためには TOPPERS Automotive Kernel で定められている型を定義したファイル「kernel.h」が必要です。またアプリケーション用の定義ファイルとして準備した「sample.h」も定義します。

```
#include "kernel.h"
#include “sample.h”
```

タスクのプロトタイプ宣言を定義します。また main()関数も本ファイルで作成するので、main()関数のプロトタイプ宣言もここで定義します。

```
void main( void );
TASK( MainTask );
```

main()関数を作成します。プログラムは起動すると、main()関数から実行されるので、ここで TOPPERS Automotive Kernel の起動を行います。StartOS()にアプリケーションモードを指定して起動します。

```
void main( void )
{
	StartOS(AppMode1);
}
```

タスクを作成します。タスクに関する具体的な定義は、SG によって生成したファイルにされているので、単純に関数を作成する手順で記述できます。

```
TASK( MainTask )
{
	/* 処理を記述 */
}
```

アプリケーションの作成手順は以上ですが、タスク内部の処理に何も記述しないのでは、タスクが正常に動作しているかわかりません。そこで TOPPERS Platform ボードに実装されている LED を点滅させます。
LED の制御をするために LED デバイスの定義ファイルをインクルードします。

```
#include “led.h”
```

LED 制御 API として以下のものが準備されています。尚、デバイスの詳細については**8章**にて触れます。

表 5.2-12 LED 制御 API

API	内容
LED_ RET LedInit(void)	LED 初期化処理
LED_ RET LedTerm(void)	LED 終了処理
LED_ RET LedOn(UINT8 led_no)	LED 点灯処理
LED_ RET LedOff(UINT8 led_no)	LED 消灯処理
LED_ RET LedRev(UINT8 led_no)	LED 点灯・消灯反転処理
UINT8 LedRef(void)	LED 点灯・消灯状態参照

TOPPERS ボードには LED が 4 つ実装されており、引数 Led_No にて所定の LED を制御します。引数 led_no に渡す値は LED2、LED3、LED4、LED5 のいずれかになります。今回は、LED2 を点滅させます。

```
TASK( MainTask )
{
	UINT32 wait_cnt = 0;

	LedInit();
	for (;;)
	{
		LedRev( LED2 );
		for( wait_cnt = 0; wait_cnt < 500000; wait_cnt++ );
	}
}
```

最終的に sample.h 及び sample.c は次のようになります

【sample.h】

```
#include "kernel_id.h"
DeclareTask( MainTask );
```

【sample.c】

```
#include "kernel.h"
#include "sample.h"
#include "led.h"

void main( void );
TASK( MainTask );

void main( void )
{
	StartOS(AppMode1);
}

TASK( MainTask )
{
	UINT32 waitCnt = 0;

	LedInit();
	for (;;)
	{
		LedRev( LED2 );
		for( waitCnt = 0; waitCnt < 500000; waitCnt++ );
	}
}
```

実際に記述したらメニューバーから「ビルド」を選び「すべてをビルド」を選択し、プログラムをビルドして下さい。正常にビルドが完了すると、**図 5.2-37**のようにビルドが正常完了したときのメッセージが表示され、TOPPERS Platform ボードに書き込むための MOT ファイル（モトローラ S レコード形式ファイル）が生成されます。

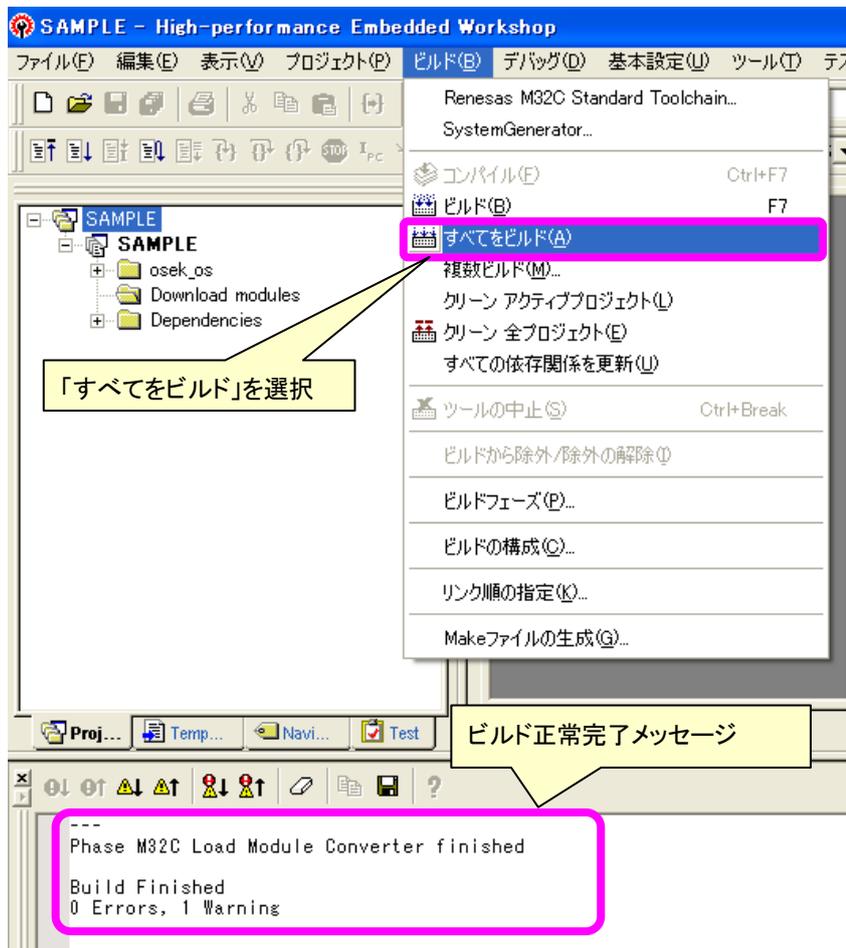
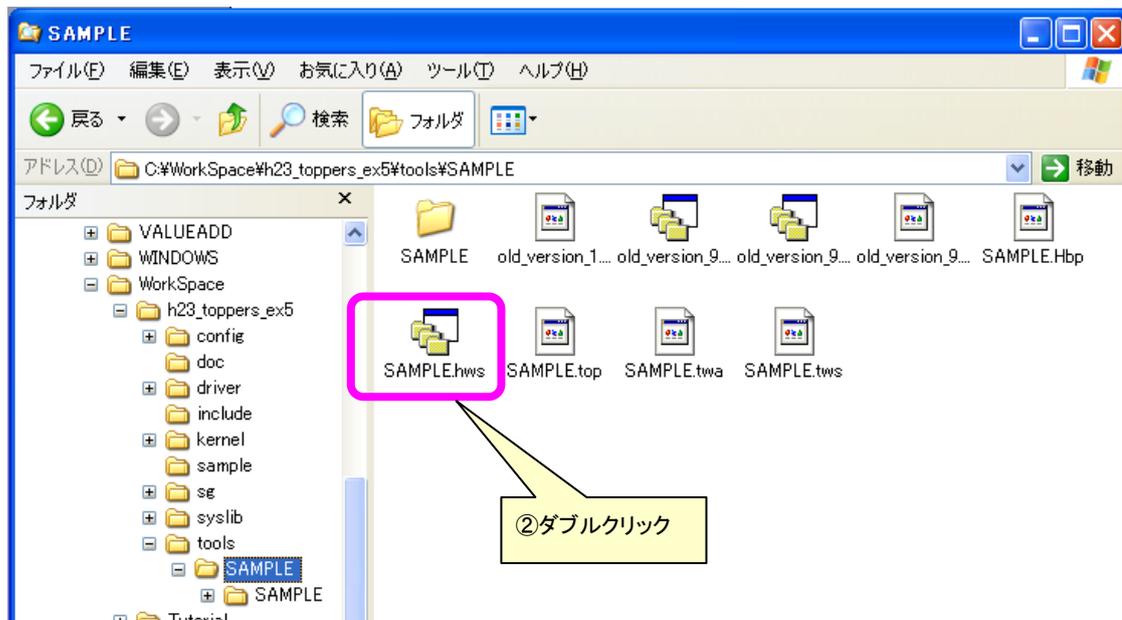
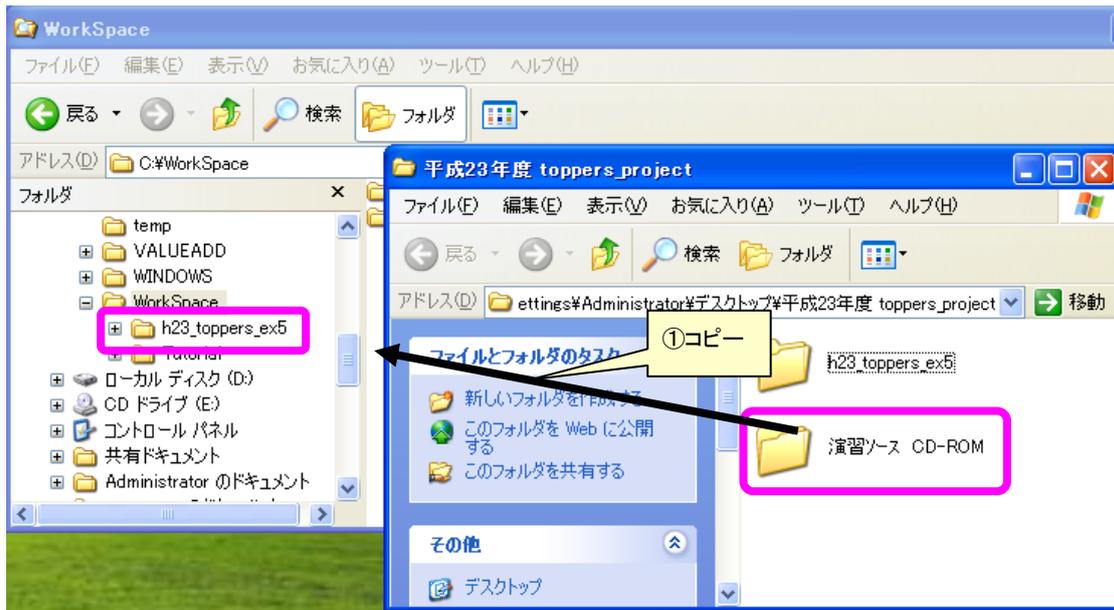



図 5.2-37 すべてをビルド

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex5」フォルダにセットアップ済みです。

① 「H25_toppers_ex5」フォルダをコピー
② 「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

5.3 プログラムの書込み

TOPPERS Platform ボードでは、ユーザ側で自由にプログラムの書き換えを行うことができるフラッシュメモリをマイコンに内蔵しています。E8a エミュレータを使用することで、USB 経由で直接フラッシュメモリにプログラムを書き込むことが可能です。本書では、E8a エミュレータを使ったプログラムの書込み例を紹介します。

また TOPPERS Platform ボードの接続方法は**図 5.3-1**を参照して下さい。

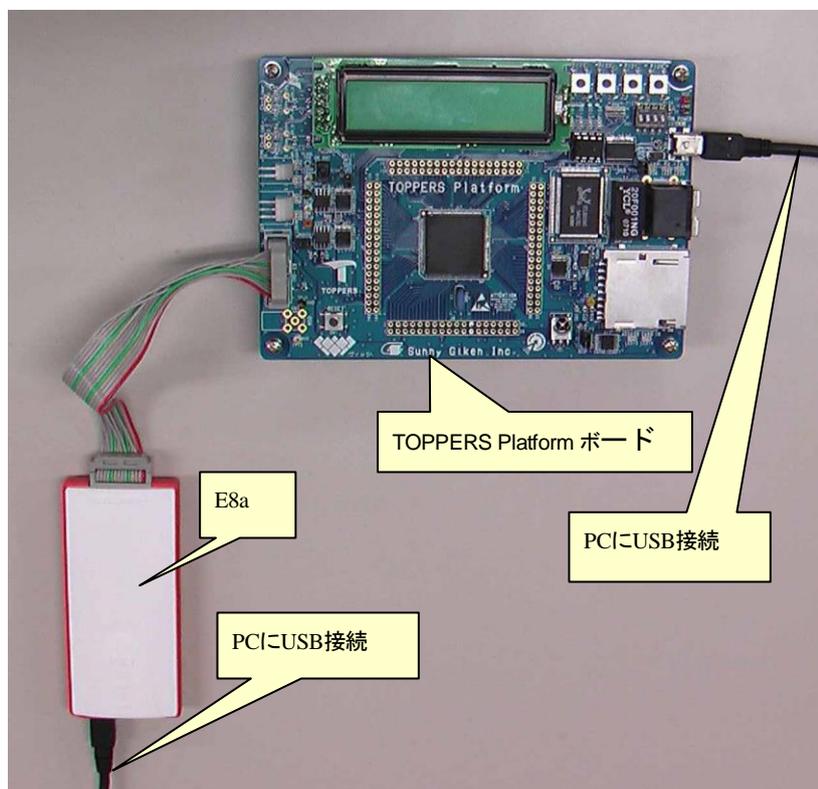



図 5.3-1 TOPPERS Platform ボード開発環境接続

5.3.1 書込み・デバッグ準備

プログラムの書込みやデバッグを行うには HEW でデバッグの設定をする必要があります。まず、HEW をスタートメニューから起動します。起動したら、メニューバーから「デバッグ」→「デバッグの設定」の順に選択します(**図 5.3-2** デバッグの設定の選択)。

図 5.3-2 デバッグの設定の選択

上記選択をすると、「デバッグの設定」ウィンドウが表示されます。左上のドロップダウンリストボックスを「All Sessions」にします。「ターゲット」タブを選択し、**表 5.3-1**の様に設定します。尚、ダウンロードモジュールは「追加」ボタンを押して表示される「ダウンロードモジュール」ウィンドウにて設定します。

表 5.3-1 新規プロジェクト作成の設定値

項目		設定値
ターゲット		M32C E8 SYSTEM
デバッグ対象フォーマット		IEEE695_RENESAS
ダウンロードモジュール 1	ファイル名	$(CONFIGDIR)¥$(PROJECTNAME).x30
	オフセット	0
	フォーマット	IEEE695_RENESAS
ダウンロードモジュール 2	ファイル名	$(CONFIGDIR)¥$(PROJECTNAME).mot
	オフセット	0
	フォーマット	S-Record

コンパイラ（nc308）が生成するオブジェクトファイルは IEEE695 フォーマットです。「デバッグ対象フォーマット」には IEEE695_RENESAS を設定します。

また、デバッグには IEEE695 フォーマットのオブジェクトファイルを用いますが、書込みにはモトローラ S-Record フォーマットのファイルを用います。そのため、ダウンロードモジュールの「フォーマット」の設定情報は、**表 5.3-1** のようになります。

設定が完了したら「OK」ボタンを押して、設定内容を確定します。（**図 5.3-3**）。

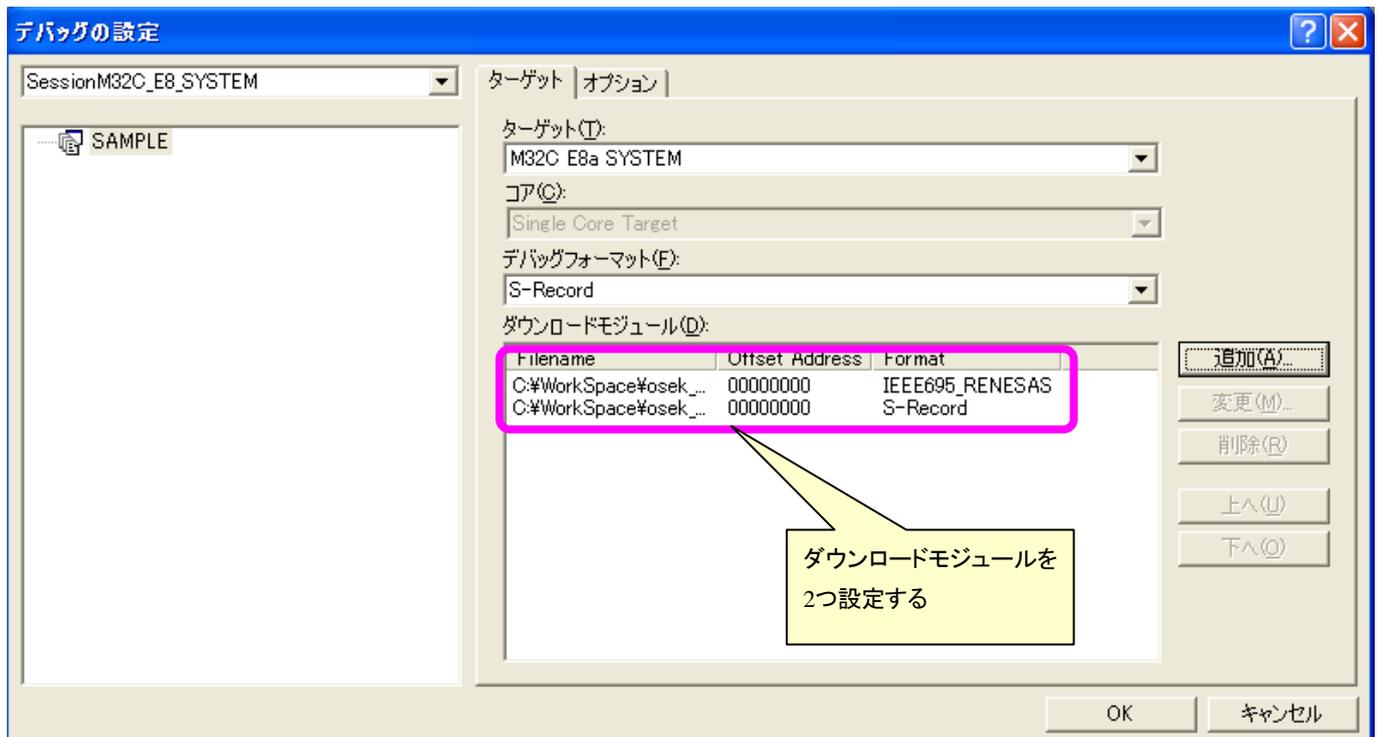



図 5.3-3 デバッグのターゲット設定

5.3.2 書込み手順

まず、ビルドの構成を選択します。メニューバーから「ビルド」→「ビルドの構成」の順に選択します（**図 5.3-4**）。

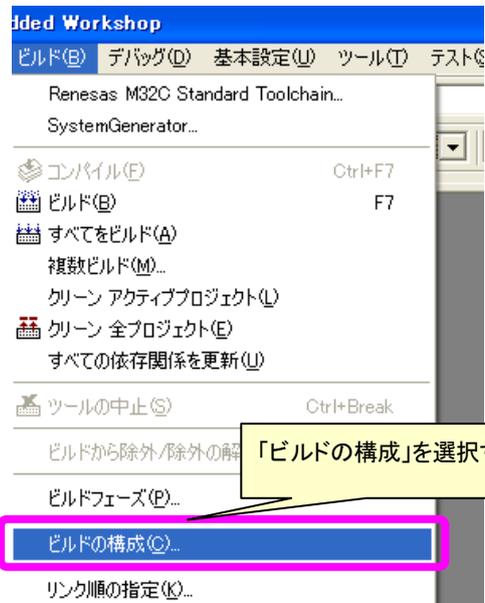



図 5.3-4 デバッグセッションの構成の選択

上記選択をすると、「ビルドコンフィグレーション」ウィンドウが表示されます。ビルドの構成を「ビルドコンフィグレーション」に前以て登録しておくことで、リリース版のビルド構成やデバッグ版のビルド構成などを容易に切り替えることができます。切り替えは「現在のコンフィギュレーション」で行います。今回は、「現在のコンフィギュレーション」を Debug_M32C_E8_SYSTEM にします。設定が完了したら「OK」ボタンを押して、設定内容を確定します（**図 5.3-5**）。

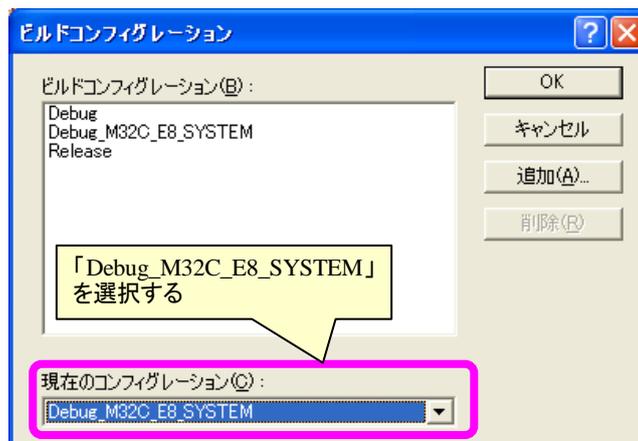



図 5.3-5 現在のコンフィグレーションの設定

次に、デバッグセッションを選択します。メニューバーから「デバッグ」→「デバッグセッション」の順に選択します(**図 5.3-6** デバッグセッションの選択)。

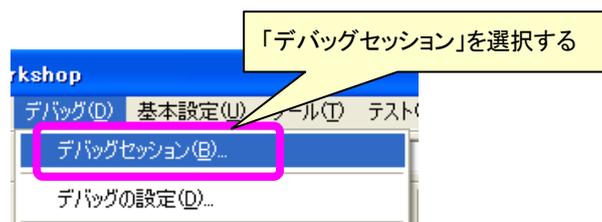



図 5.3-6 デバッグセッションの選択

上記選択をすると、「デバッグセッション」ウィンドウが表示されます。「デバッグセッション」に前以て登録しておくことで、デバッグセッションを容易に切り替えることができます。切り替えは「現在のセッション」で行います。今回は、「現在の現在のセッション」を「SessionM32C_E8_SYSTEM」にします。設定が完了したら

「OK」ボタンを押して、設定内容を確定します（**図 5.3-7**）。

図 5.3-7 デバッグセッションの設定

次に、接続を選択します。メニューバーから「デバッグ」→「接続」の順に選択します(**図 5.3-8** 接続の選択)。

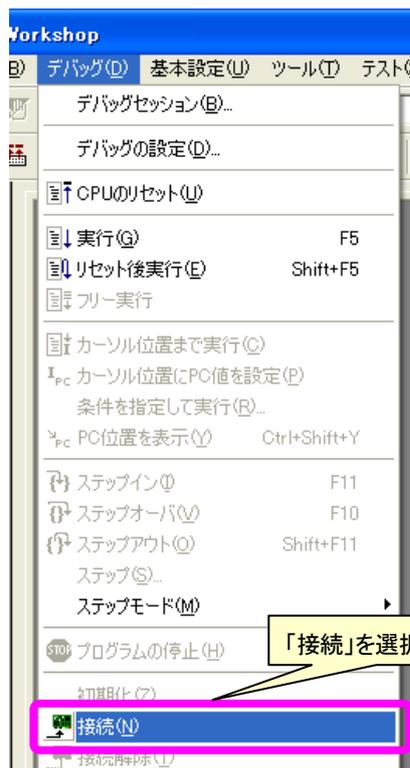



図 5.3-8 接続の選択

上記選択をすると、「エミュレータ設定」ウィンドウが表示されます。「エミュレータモード」タブを選択し、**表 5.3-2**の様に設定します。

表 5.3-2 エミュレータモード設定

項目	設定値
MCU グループ	M32C/85 Group
デバイス	M30855FJ
モード	フラッシュメモリデータの書込み

エミュレータから電源供給	チェックしない

今回、E8a エミュレータをフラッシュメモリのライタとして使用するので「フラッシュメモリデータの書込み」を選択します。また TOPPERS Platform ボードの電源を別途 USB から供給している場合、「エミュレータから電源供給」にチェックは不要です。設定が完了したら「OK」ボタンを押します（**図 5.3-9**）。

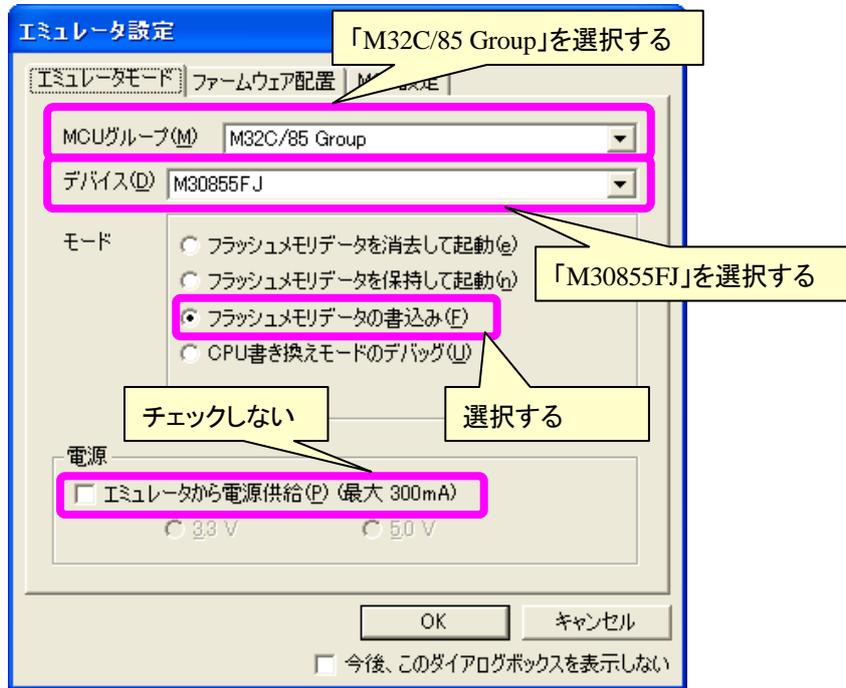



図 5.3-9 エミュレータモードの設定

「ID コード確認」ウィンドウが表示されます。フラッシュメモリに書き込まれている ID コードと ID コードの入力モードの設定を行います。設定が完了したら「OK」ボタンを押します。「debugger」ウィンドウが 2 回表示されますが、どちらも設定項目はないので「OK」ボタンを押して閉じます（**図 5.3-10**）。

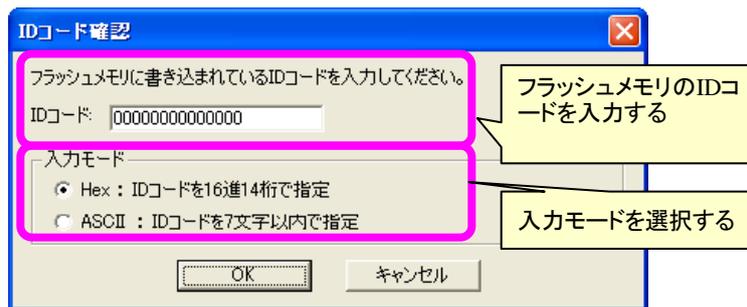



図 5.3-10 ID コード確認画面

次に、ツリーから「SAMPLE.mot – 00000000」を右クリックし、「ダウンロード」を選択すると、フラッシュメモリへプログラムの書込みが始まります（**図 5.3-11**）。しばらくして「debugger」ウィンドウが表示されれば、書込みが完了します。「OK」ボタンを押して閉じます。

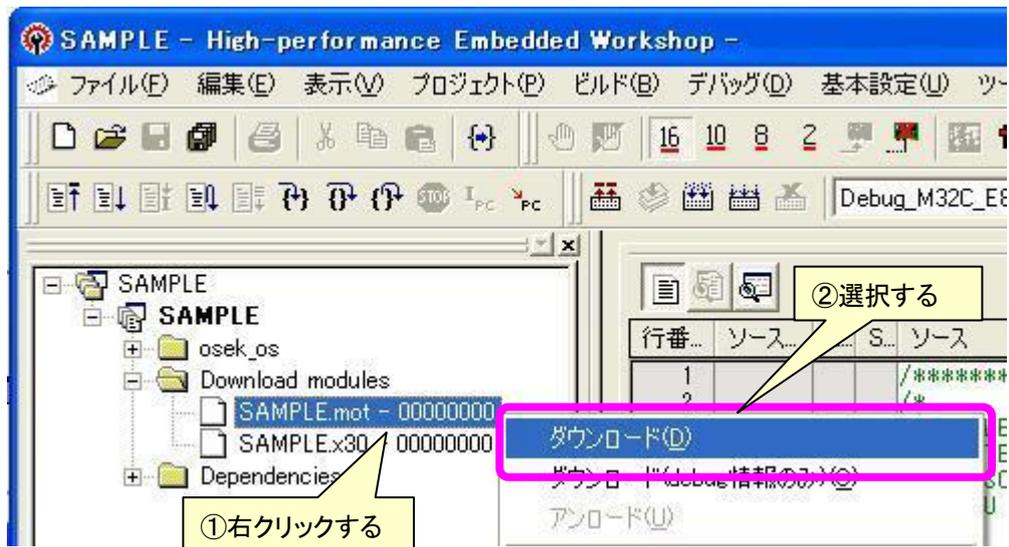



図 5.3-11 ダウンロードの選択

次に、メニューバーから「デバッグ」→「接続解除」の順に選択し、E8a エミュレータとの接続を解除します（**図 5.3-12**）。

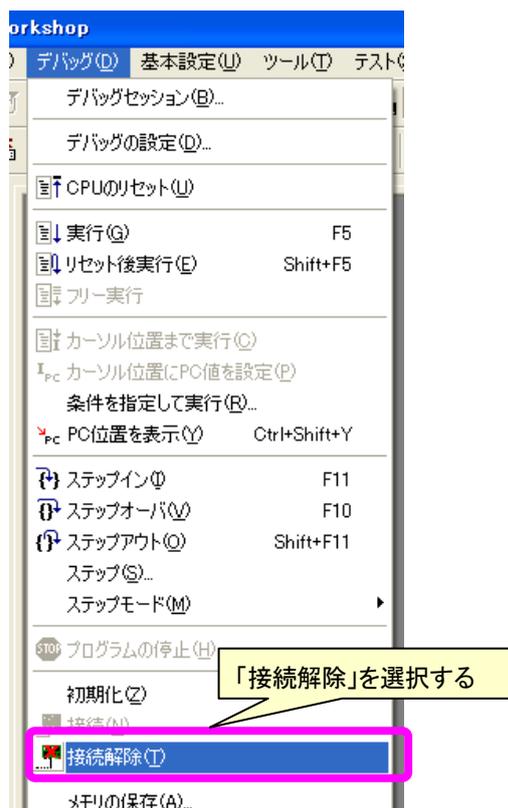



図 5.3-12 接続解除の選択

E8a エミュレータを TOPPERS Platform ボードから外して、ボード上でリセットを行うと、書き込んだプログラムが実行されます。

5.3.3 デバッグ手順

はじめの手順は**5.3.2 書込み手順**と同じです。**図 5.3-4**～**図 5.3-8**までの設定を参考にし、「エミュレータ設定」ウィンドウを表示します。「エミュレータモード」タブを選択し、**表 5.3-3**の様に設定します（**図 5.3-13**）。

表 5.3-3 エミュレータモード設定

項目	設定値
MCU グループ	M32C/85 Group
デバイス	M30855FJ
モード	フラッシュメモリデータを消去して起動
エミュレータから電源供給	チェックしない

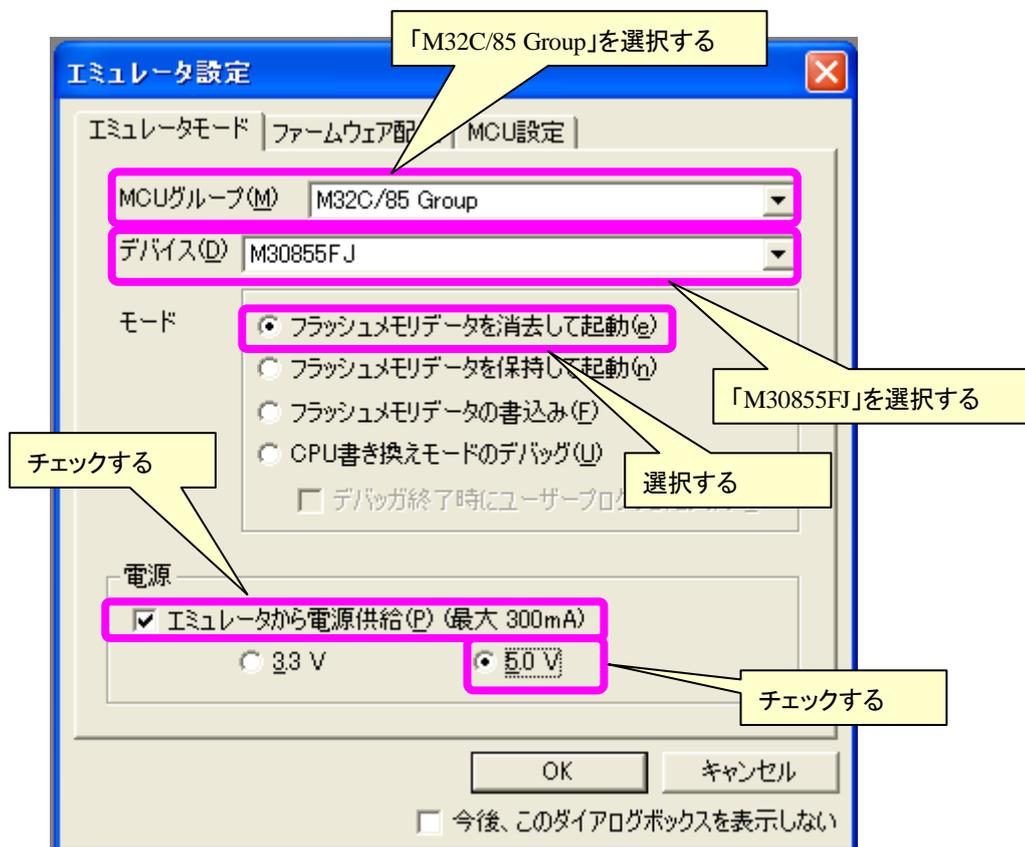



図 5.3-13 エミュレータモードの設定

「ファームウェア配置」タブを選択し、**表 5.3-4**の様に設定します。設定が完了したら「OK」ボタンを押します（**図 5.3-14**）。

表 5.3-4 ファームウェア配置設定

項目	設定値
プログラム	FFF0
ワーク RAM	60

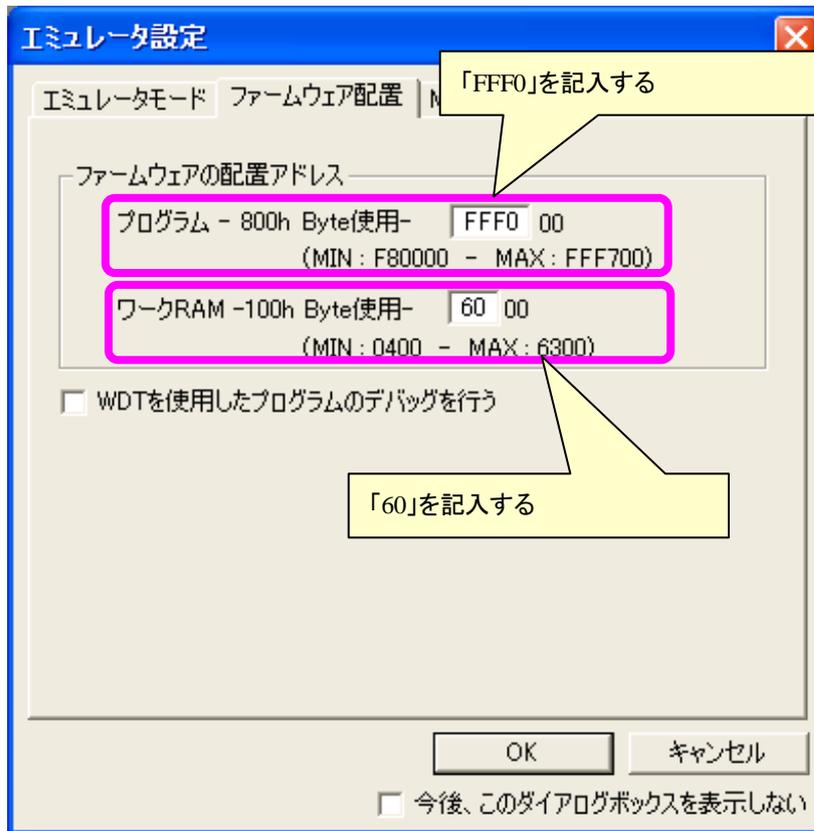



図 5.3-14 ファームウェア配置の設定

次に、ツリーから「SAMPLE.x30 – 00000000」を右クリックし、「ダウンロード」を選択すると、フラッシュメモリへプログラムの書込みが始まります（**図 5.3-15**）。

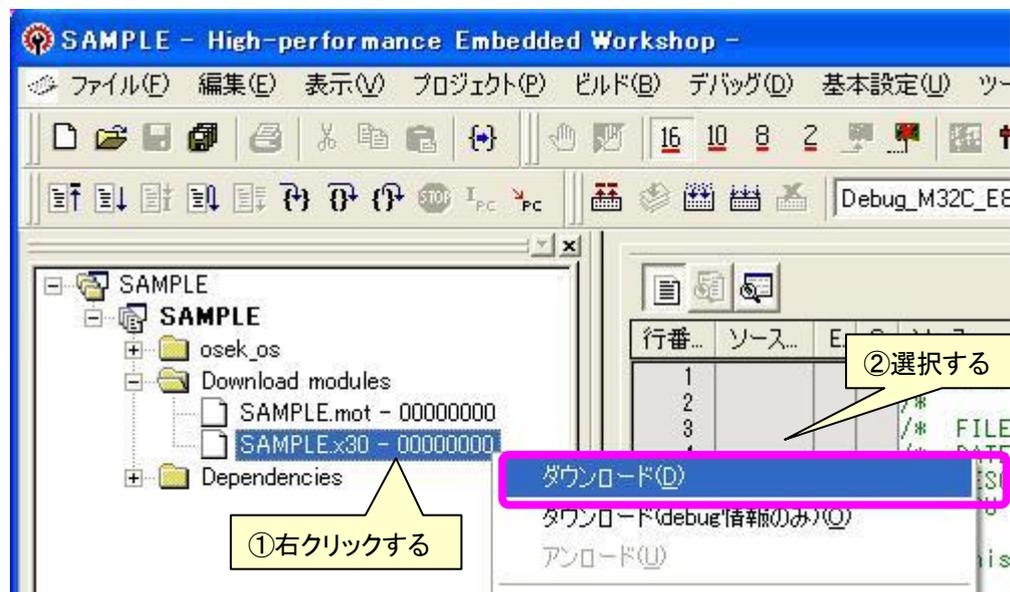



図 5.3-15 ダウンロードの選択

しばらくすると Hew の「Debug」タブに**図 5.3-16**のようなメッセージが表示され、デバッグできる状態になります。

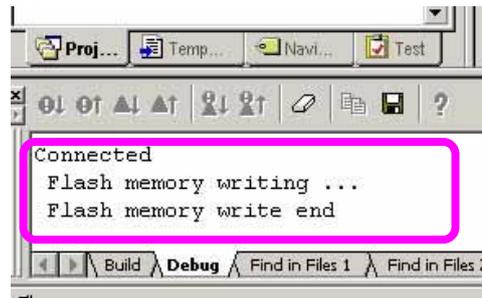



図 5.3-16 フラッシュメモリ書込み完了メッセージ

Hew の「S/W ブレークポイント」上でダブルクリックするとブレークポイントを設定できます。プログラムの実行を停止させたい行にブレークポイントを設定します（**図 5.3-17**）。

ダブルクリックして、
ブレークポイント(ハードウェアブレーク)を設定

行番...	ソースアドレス	E...	S/Wブレ...	ソース
33				/* 戻り値 : 無し */
34				/* 引数 : 無し */
35				/***/
36				void main(void)
37	F81CC6			{
38	F81CC6			StartOS(AppMode1);
39	F81CCC			}
40				
41				/***/
42				/* 関数名 : TASK(MainTask) */
43				/* 役割 : メインタスク */
44				/* 戻り値 : 無し */
45				/* 引数 : 無し */
46				/***/
47				TASK(MainTask)
48	F81CCE			{
49	F81CD0			UINT32 waitCnt = 0;
50				
51	F81CD4			LedInit();
52				for (;;)
53	F81CD8			{
54	F81CD8	●	⇨	LedRev(LED2);
55	F81CDE			for(waitCnt = 0; waitCnt < 500000; waitCnt++);
56				}
57				}
58				

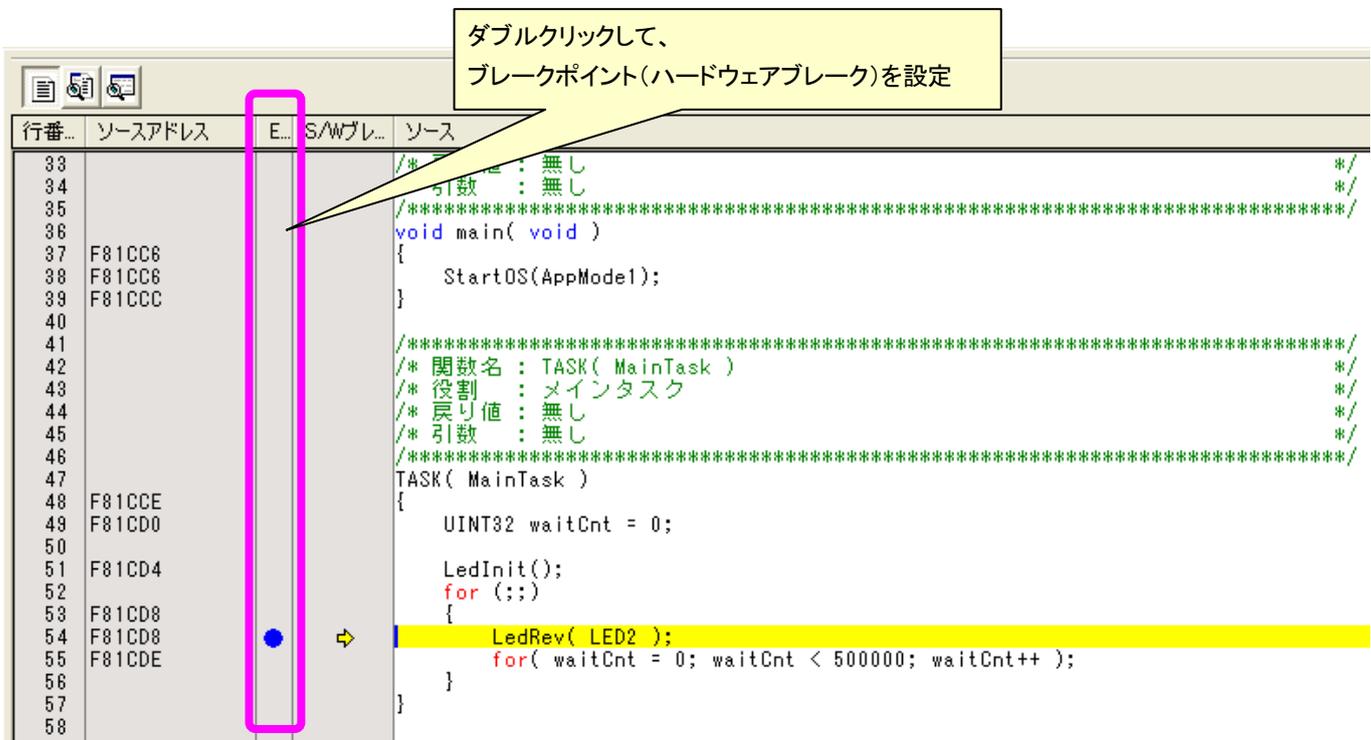



図 5.3-17 ブレークポイントの設定

メニューバーから「デバッグ」→「リセット実行」の順に選択します。ブレークポイントを設定した位置まで実行します（**図 5.3-18**）。

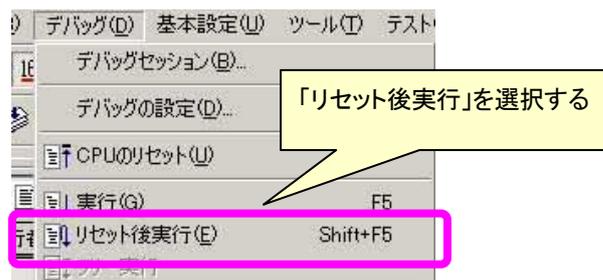



図 5.3-18 リセット後実行

これでデバッグの確認作業は終了です。

ブレークポイントの設定にはふたつの方法があります。

■　ソフトウェアブレーク

ソフトウェアブレークは、ブレークしたい番地の命令を停止命令に書き換えることで実現します。当該番地がＲＡＭ上にある場合は問題ありませんが、フラッシュメモリ上にある場合はフラッシュメモリ書き換え機能を使ってフラッシュメモリ上の命令を停止命令に書き換えます。

フラッシュメモリには最大書き込み回数という制限があります。したがって、頻繁にソフトウェアブレークを使用するのはマイコンの信頼性という観点からあまりお勧めはできません。

■　ハードウェアブレイク

ハードウェアブレークはマイコン内蔵の H-UDI（ヒューマン・ユーザーデバッグインタファフェース：内蔵デバッグ機能）を使用します。H-UDI はブレーク設定した命令の番地を常に監視しており、設定番地とメモリアクセスアドレスが一致するとプログラムを停止させます。

マルチタスクプログラミング

6.1 タスクの作成

タスクは、下記の２点を除いてＣ言語の関数作成と変わりません。

- 関数名は、「TASK(タスク名) 」にすること
- タスクの外部宣言は「DeclareTask（タスク名）」にすること

作成するタスクは、実行方法によって２種類に分類されます。

■ １度だけ処理を実行する場合

Ｃ言語の関数のように、関数が呼び出された際に１度だけ実行する場合は下記の様に記述します。起床したタスクは、実行状態に移った際に、記述した処理を行って終了します。このとき、TASK の最後に「TerminateTask」を呼び、自タスクの終了を行う必要があります。

```
TASK(exsample)
{
    /* ここに処理を記述します */

    TerminateTask ();
}
```

■ 処理を永続的に実行する場合

組込みシステムの場合、タスクは永続的に処理し続けるのが一般的です。この場合、for 文や while 文を用いて無限ループを作成して、タスクが終了しないようにして下さい。下記に作成例を示します。

```
TASK(exsample)
{
    for (;;)
    {
        /* ここに処理を記述します */
    }
}
```

6.2 タスク制御

タスクは、大きく分けて下記の状態に遷移されます。

- 実行状態　　　…　処理がマイコンにより実行されている状態
- 実行可能状態　…　マイコンが他の処理を行っているため、実行待ちをしている状態
- 休止状態　　　…　タスクが起動されていない状態
- 待ち状態　　　…　１つ以上のイベントを待っている状態

つまり、マイコンによって処理が実行されている状態を「実行状態」といい、処理は実行可能ですが、他の処

理を実行しているために処理が実行出来ない状態（実行を準備している状態）にあることを「実行可能状態」といいます。またマイコンにより処理が実行されていない状態を「休止状態」といい、イベントにより同期待ちしており、解除されるまで処理が行えない（停止）状態を「待ち状態」といいます。
　待ち状態になれるのは拡張タスクだけで、基本タスクは待ち状態になることができません。その代わり、基本タスクは拡張タスクと比較してメモリ使用量を削減した実装が可能です。

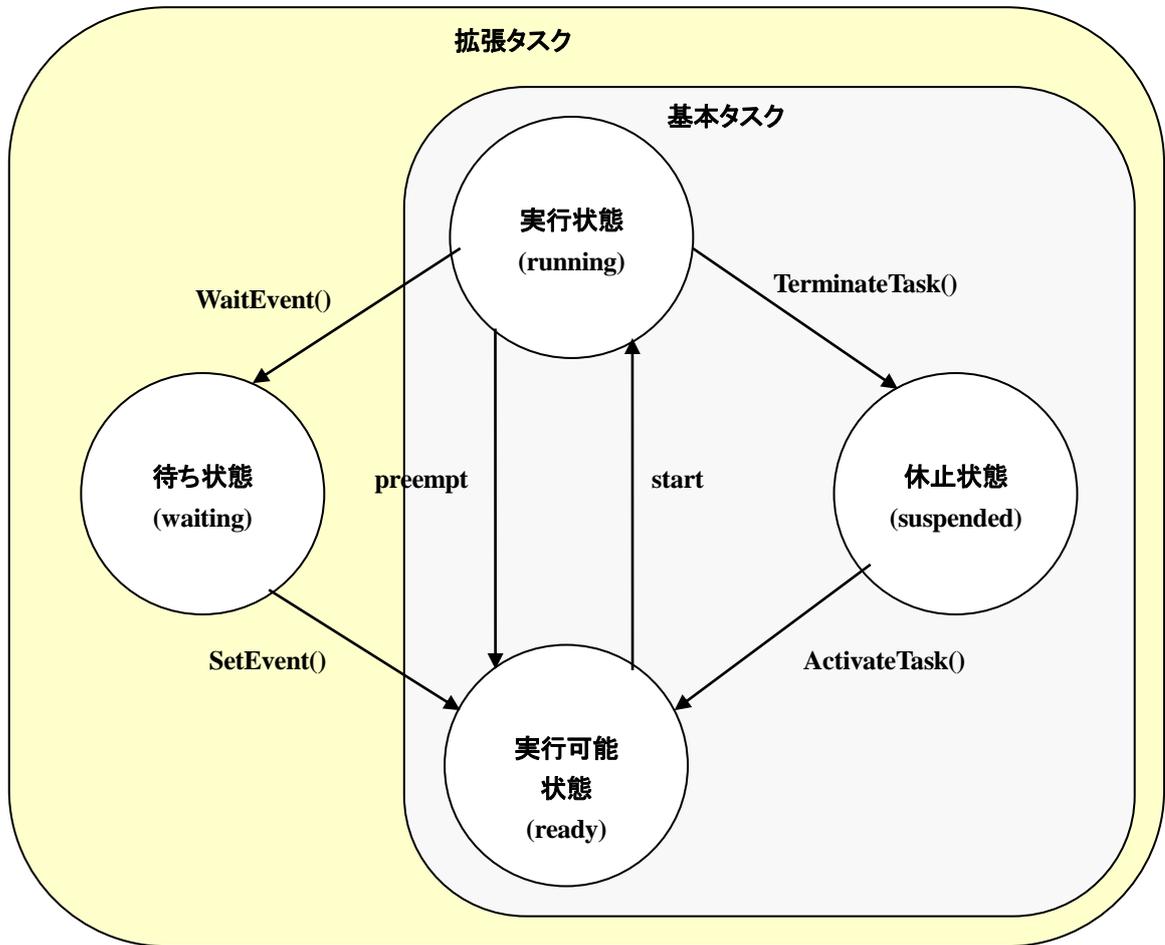



図 6.2-1　4 状態の状態遷移図

■　タスクの宣言

　タスクを使用するには、OIL ファイルでタスクを定義することが必要です。

```
TASK Task1 {                          /* タスク名 */
        PRIORITY = 5;                 /* タスク優先度 */
        ACTIVATION = 1;               /* 起動要求キューイング数 */
        SCHEDUKE = FULL;              /* スケジューリングポリシー */
        STACKSIZE = 0x200;            /* スタックサイズ */
};
```

■　タスクの開始

　タスクを開始するには、プログラム中で ActivateTask という API を使用します。タスクは実行可能状態に遷移し、動作可能な状態となります。

```
ActivateTask(Task1);          /* Task1 の起動 */
```

■　**状態遷移サンプル１**

　具体的なサンプルプログラムを基に状態遷移の例を示します。次のプログラムが実行していることを想定し、実行状態、実行可能状態、休止状態の遷移について考えます。

【sample.oil】

```
#include "implementation.oil"

CPU current {
#include <t100us_timer.oil>

	OS os {
		STATUS = STANDARD;
		STARTUPHOOK = TRUE;
		ERRORHOOK = FALSE;
		SHUTDOWNHOOK = FALSE;
		PRETASKHOOK = FALSE;
		POSTTASKHOOK = FALSE;
		USEGETSERVICEID = TRUE;
		USEPARAMETERACCESS = TRUE;
		USERESSCHEDULER = FALSE;
	};

	APPMODE AppMode1 {};

	TASK Led2Task {
		AUTOSTART = TRUE {
			APPMODE = AppMode1;
		};
		PRIORITY = 13;
		STACKSIZE = 0x0100;
		ACTIVATION = 1;
		SCHEDULE = FULL;
	};

	TASK Led3Task {
		AUTOSTART = FALSE;
		PRIORITY = 14;
		STACKSIZE = 0x0100;
		ACTIVATION = 1;
		SCHEDULE = FULL;
	};
};
```

【sample.h】

```
#include "kernel_id.h"
DeclareTask( Led2Task );
DeclareTask( Led3Task );
```

【sample.c】

```
#include "kernel.h"
#include "sample.h"
#include "led.h"

#define WAIT_CNT ((UINT32)500000ul)

void main( void );
TASK( Led2Task );
TASK( Led3Task );

void main( void )
{
	StartOS(AppMode1);					/* OS スタート			*/
}

TASK( Led2Task )
{
	volatile UINT32 wait_cnt = 0;

	while(1) {
		LedRev( LED2 );					/* LED2 点灯/消灯		*/
		for( wait_cnt = 0; wait_cnt < WAIT_CNT; wait_cnt++ );	/*  一定時間待つ		*/
		ActivateTask( Led3Task );			/* LED3 タスクの起動		*/
	}
	TerminateTask();					/*  自タスク終了		*/
}

TASK( Led3Task )
{
	volatile UINT32 wait_cnt = 0;

	LedOn( LED3 );						/* LED3 点灯
	  */
	for( wait_cnt = 0; wait_cnt < WAIT_CNT; wait_cnt++ );	/*  一定時間待つ		*/
	LedOff( LED3 );						/* LED3 消灯			*/
	TerminateTask();					/*  自タスク終了		*/
}

#ifdef USE_STARTUPHOOK
void StartupHook( void )
{
	LedInit();						/* LED 初期化			*/
}	/* StartupHook		*/
#endif /* USE_STARTUPHOOK */
```

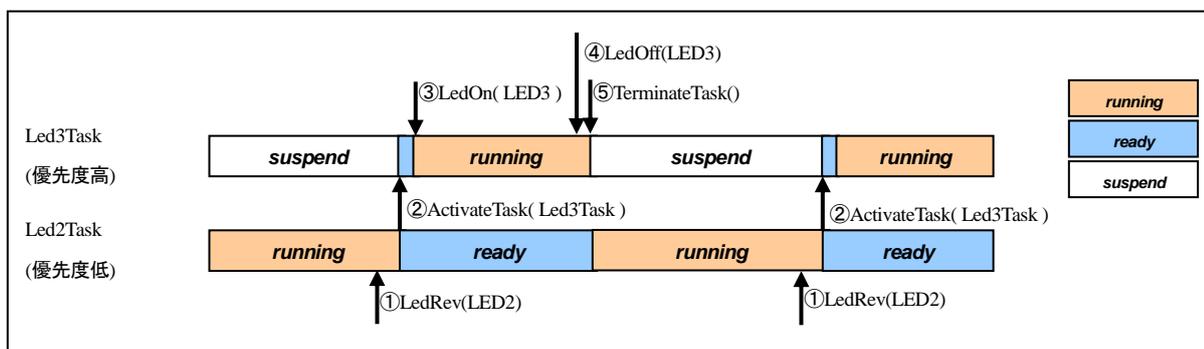



図 6.2-2 実行状態、実行可能状態、休止状態の遷移

Led2Task は LED2 を制御するタスクです。LED2 の点灯、消灯と Led3Task の実行を行います。Led3Task は自タスクが実行状態のときだけ LED3 を点灯します。また Led2Task より Led3Task の優先度は高く、両タスクともフルプリエンプティブです。

① Led2Task が実行状態のときに LED2 が点灯と消灯を反転します。尚、Led2Task は sample.oil にて TOPPERS Automotive Kernel 起動時に自動起動する設定になっています。
② Led2Task から Led3Task を実行します。Led3Task は休止状態から実行可能状態に遷移します。さらに Led2Task より優先度が高く、フルプリエンプティブなのですぐ実行状態に遷移します。Led2Task は Led3Task が実行状態に遷移したことで、実行状態から実行可能状態に遷移します。
③ Led3Task が実行状態に遷移した直後に LED3 を点灯します。
④ Led3Task の終了直前に LED3 を消灯します。
⑤ Led3Task は自分自身を終了し、休止状態に遷移します。Led3Task が休止状態になると実行可能状態に遷移していた Led2Task が実行状態になります。

以降、①～⑤の動作を繰返します。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex6_act_ter」フォルダにセットアップ済みです。

①「H25_toppers_ ex6_act_ter 」フォルダをコピー
②「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

■ **イベントの宣言とタスクとの関連付け**

イベントも OIL ファイルで定義します。また、イベント待ちを行うタスクとイベントを関連付けます。

```
TASK Task1 {                      /* タスク名 */
        PRIORITY = 5;             /* タスク優先度 */
        ACTIVATION = 1;           /* 起動要求キューイング数 */
        SCHEDUKE = FULL;          /* スケジューリングポリシー */
        STACKSIZE = 0x200;        /* スタックサイズ */
        EVENT= Event1;            /* 使用するイベント */
};

TASK Task2 {                      /* タスク名 */
        PRIORITY = 3;             /* タスク優先度 */
        ACTIVATION = 1;           /* 起動要求キューイング数 */
        SCHEDUKE = FULL;          /* スケジューリングポリシー */
        STACKSIZE = 0x200;        /* スタックサイズ */
};

EVENT Event1 {                    /* イベント名 */
        MASK = AUTO;              /* イベントマスク */
};
```

■ イベント待ち、設定、クリア

イベント待ちするには、プログラム中で WaitEvent という API を使用します。またイベント設定は SetEvent、イベントクリアは ClearEvent を使用します。

```
TASK( Task1 )
{
        WaitEvent( Event1 );                    /* イベント待ち          */
        ClearEvent( Event1 );                   /* イベントクリア        */
}

TASK( Task2 )
{
        SetEvent( Task1, Event1 );              /* イベント設定          */
}
```

■ 状態遷移サンプル 2

次のプログラムが実行していることを想定し、実行状態、実行状態、待ち状態の遷移について考えます。
尚、プログラム内で使用している LCD 制御 API の機能は**表 6.2-1**を参照して下さい。デバイスの詳細については**8.1デバイスドライバ**にて触れます。

表 6.2-1 LCD 制御 API

API	内容
LCD_RET LcdInit(void)	LCD ドライバ初期化処理。
LCD_RET LcdCtlDisplay(LCD_CH_NO ch_no, LCD_CTL_CODE ctl_code)	LCD デバイス表示制御処理。ctl_code に LCD_CTL_CLRDISPLAY を設定すると表示をクリアする。
LCD_RET LcdWriteLine(LCD_CH_NO ch_no, LCD_CHARACTER *row, UINT8 line)	LCD 一行データ書込み処理。line で書込む行を指定し、書込む文字列を row に設定する。

【sample.oil】

```
#include "implementation.oil"

CPU current {
#include <t100us_timer.oil>

	OS os {
		STATUS = STANDARD;
		STARTUPHOOK = TRUE;
		ERRORHOOK = FALSE;
		SHUTDOWNHOOK = FALSE;
		PRETASKHOOK = FALSE;
		POSTTASKHOOK = FALSE;
		USEGETSERVICEID = TRUE;
		USEPARAMETERACCESS = TRUE;
		USERESSCHEDULER = FALSE;
	};

	APPMODE AppMode1 {};

	TASK LcdTask {
		AUTOSTART = TRUE {
			APPMODE = AppMode1;
		};

		PRIORITY = 14;
		STACKSIZE = 0x0100;
		ACTIVATION = 1;
		SCHEDULE = FULL;
		EVENT = LedEvt;
	};

	TASK LedTask {
		AUTOSTART = TRUE {
			APPMODE = AppMode1;
		};
		PRIORITY = 13;
		STACKSIZE = 0x0100;
		ACTIVATION = 1;
		SCHEDULE = FULL;
	};

	EVENT LedEvt {
		MASK = AUTO;
	};
};
```

【sample.h】

```
#include "kernel_id.h"
DeclareEvent( LedEvt );
DeclareTask( LcdTask );
DeclareTask( LedTask );
```

【sample.c】

```
#include "kernel.h"
#include "sample.h"
#include "led.h"
#include "serial.h"
#include "lcd.h"

#define WAIT_CNT ((UINT32)5000000ul)

static UINT8 lcd_str[LCD_DEV_LINE][LCD_DEV_DIGIT+1]= {
		"LED2 ON ",
		"LED2 OFF",
};

void main( void );
TASK( LcdTask );
TASK( LedTask );

void main( void )
{
		StartOS(AppMode1);					/* OS スタート			*/
}

TASK( LcdTask )
{
		while( 1 ) {
				if(LedRef() & LED2){			/* LED2 の状態参照		*/
						LCDWriteLine(0, lcd_str[0], 1);	/*  点灯時メッセージ表示	*/
				}
				else {
						LCDWriteLine(0, lcd_str[1], 1);	/*  消灯時メッセージ表示	*/
				}
				WaitEvent( LedEvt );			/*  イベント待ち		*/
				ClearEvent( LedEvt );			/*  イベントクリア		*/
		}
		TerminateTask();				/*  自タスク終了		*/
}

TASK( LedTask )
{
		volatile UINT32 wait_cnt = 0;

		while( 1 ) {
				LedRev( LED2 );				/* LED2 点灯/消灯		*/
				SetEvent( LcdTask, LedEvt );		/*  イベント設定		*/
				for( wait_cnt = 0; wait_cnt < WAIT_CNT; wait_cnt++ );	/*  一定時間待つ	*/
		}
		TerminateTask();				/*  自タスク終了		*/
}

#ifdef USE_STARTUPHOOK
void StartupHook( void )
{
		/* LED 初期化		*/
		LedInit();

		/* LCD 初期化  */
		LCDInit();

		/* LCD クリア  */
		LCDCtlDisplay(0, LCD_CTL_CLRDISPLAY);
}		/* StartupHook		*/
#endif /* USE_STARTUPHOOK */
```

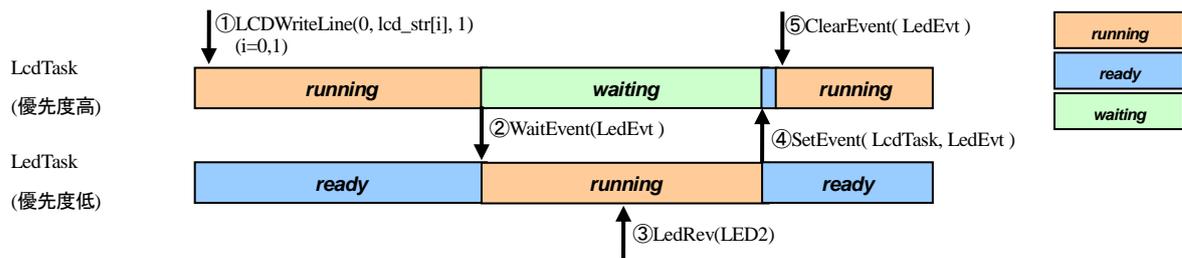



図 6.2-3 実行状態、実行可能状態、待ち状態の遷移

LedTask は LED2 を制御するタスクです。LED2 の点灯、消灯とイベントの設定を行います。LcdTask は LED2 の状態を LCD に表示します。普段はイベント待ちしており、LED2 の状態が変化したときだけ実行します。また LedTask より LcdTask の優先度は高く、両タスクともフルプリエンプティブです。

① LcdTask が実行状態のときに LCD に LED の状態を表示します。尚、LedTask と LcdTask は sample.oil にて TOPPERS Automotive Kernel 起動時に自動起動する設定になっています。
② LcdTask は自分自身を待ち状態に遷移します。また LcdTask が待ち状態になったことで LedTask が実行可能状態から実行状態に遷移します。
③ LedTask は LED2 の点灯と消灯を反転します。
④ LED の状態が変化したことを LcdTask に知らせるため、LedTask はイベントを設定します。イベントが設定されたことで LcdTask は待ち状態から実行可能状態に遷移します。さらに LedTask より優先度が高く、フルプリエンプティブなのですぐ実行状態に遷移します。LcdTask が実行状態に遷移したことで、LedTask は実行状態から実行可能状態に遷移します。
⑤ LcdTask はイベントをクリアします。

以降、①~⑤の動作を繰返します。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex6_eve」フォルダにセットアップ済みです。

①「H25_toppers_ ex6_eve 」フォルダをコピー
②「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

6.3 アラーム機能

アラーム機能は、時間の経過をきっかけに特定の処理を実行する機能で、アラームとカウンタにより構成されます。カウンタは時間やエンジンクランク角など一定間隔の信号のベースとしカウントアップします。アラームはカウンタが一定値に達した（満了）場合にアクションを起こします。

アラームには次のような設定が可能です。

- カウンタ値の設定方法　…　絶対アラーム、相対アラーム
- アラームの設定方法　…　シグナルアラーム、周期アラーム
- 実行可能なアクション　…　タスク起動、イベント設定、コールバックルーチンの実行

絶対アラームはカウンタが設定した値になった場合にカウンタが満了したと判断します。相対アラームはアラームを実行してからカウンタがカウントアップした値が設定した値になった場合に満了したと判断します。

またアラームを実行し１度だけ処理を行うものがシングルアラームで、周期ごとに繰返し処理を行うものが周期アラームです。カウンタ満了時にはタスク起動、イベント設定、コールバックルーチンのいずれかのアクションを実行することが可能です。

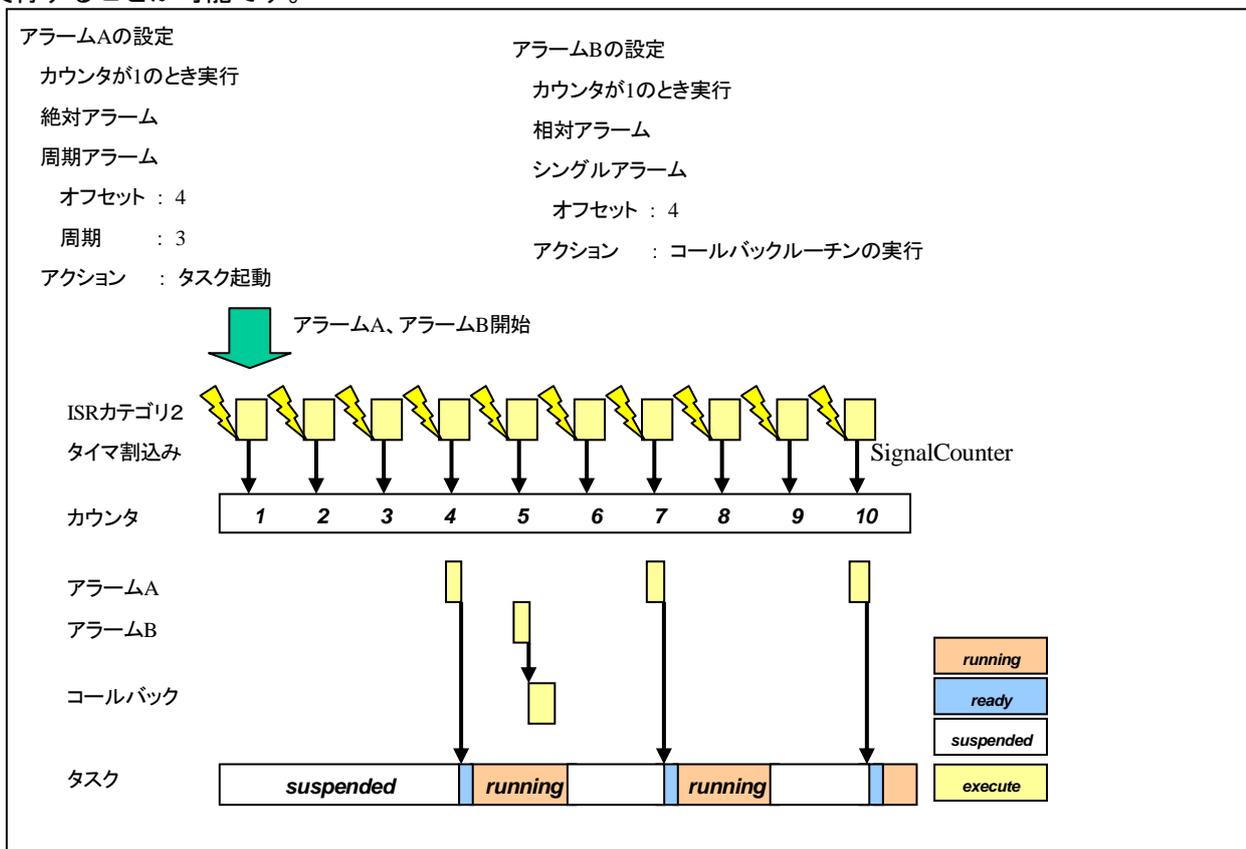



図 6.3-1 アラームの動作

図 6.3-1は、アラームＡとアラームＢをカウンタが１のときに開始した場合の動作を示しています。

アラームＡは絶対アラームなので、カウンタが４になったときに初めて満了し、タスクを起動します。また周期アラームを設定しているため、それ以降は３周期ごとに満了し、タスクを起動します。

アラームＢは相対アラームなので、オフセットとアラーム開始時のカウンタを合計した５のときに満了し、コールバックルーチンの実行を行います。シングルアラームのため、それ以降はコールバックルーチンの実行は行われません。

■ カウンタとアラームの宣言

カウンタとアラームも OIL ファイルで定義します。アラームを駆動するカウンタを関連付けます。

```
TASK Task1 {                            /* タスク名 */
        PRIORITY = 5;                   /* タスク優先度 */
        ACTIVATION = 1;                 /* 起動要求キューイング数 */
        SCHEDUKE = FULL;                /* スケジューリングポリシー */
        STACKSIZE = 0x200;              /* スタックサイズ */
};

COUNTER Counter1 {                      /* カウンタ名 */
        MINCYCLE = 10;                  /* 最小の周期 */
        MAXALLOWEDVALUE = 99;           /* カウンタ最大値 */
        TICKSPERBASE = 10;              /* 1 回のカウンタ増加要求で加算されるティック数 */
}

ALARM Alarm1 {                  /* アラーム名 */
        COUNTER = Counter1;             /* 駆動するカウンタ */
        ACTION   = ACTIVATETASK {       /* アクションの設定 */
                TASK = Task1;
        };
};
```

また、100μs ごとに 1 回カウントアップする 100μs タイマカウンタを syslib フォルダ(5.1章参照)に準備しています[22]。100μs タイマカウンタを使用すると 1ms 周期ごとにアラームを満了するなど時間と関連付けることができます。

```
#include < t100us_timer.oil>                /* 100μs タイマのインクルード */

ALARM Alarm2 {                              /* アラーム名 */
        COUNTER = T100usTimerCnt;           /* システムタイマカウンタ */
        ACTION   = ACTIVATETASK {           /* アクションの設定 */
                TASK = Task1;
        };
};
```

■ アラームの開始

```
SetAbsAlarm(Alarm1 , 7 , 0);      /* アラームを絶対時間 7 で実行、リピートなし */
SetRelAlarm(Alarm1 , 5 , 4);      /* アラームを相対時間 5 で実行、4 ティックに一度実行 */
```

演習問題

LED を点灯させるタスクと、消灯させるタスクを 2 種類作成し、300ms 周期ごとに点灯と消灯を繰り返すシステムを作成して下さい。なお、シングルアラームと周期アラームを 1 つずつ使用すること。

[22] 100μs タイマカウンタは、TOPPERS プロジェクトで公開している TOPPERS Automotive Kernel には準備されていませんので注意してください。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex6_alm」フォルダにセットアップ済みです。

①「H25_toppers_ ex6_alm 」フォルダをコピー
②「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

6.4 排他制御

排他とは、自分や仲間以外を拒むことをいいます。排他制御とは、この排他が正常に行われる様に制御することです。排他制御を行う目的は、資源の同時利用を避けるためです。なぜ、資源の同時利用を避ける必要があるのか電車を例に説明します。

電車の線路は、一般的に「上り線」と「下り線」で一本ずつありますが、ある場所では線路が上下線共通で使用されていたとしましょう。もし、この共通で使用している線路に互いに電車が来た場合、電車は衝突する可能性があります。これでは危険なので、信号機を設けます。信号機は、上下線の共通部分に入る線路の手前に設置され、どちらか一方が共通の線路を使用している間は、信号が赤になり進入を禁止します。共通部分を通過すると、再び使用できるようになり、赤信号で待っていた電車が走り出します。これにより、共通部分の線路間でも安全に使用することができます。これが、排他制御の基本的な考えです。

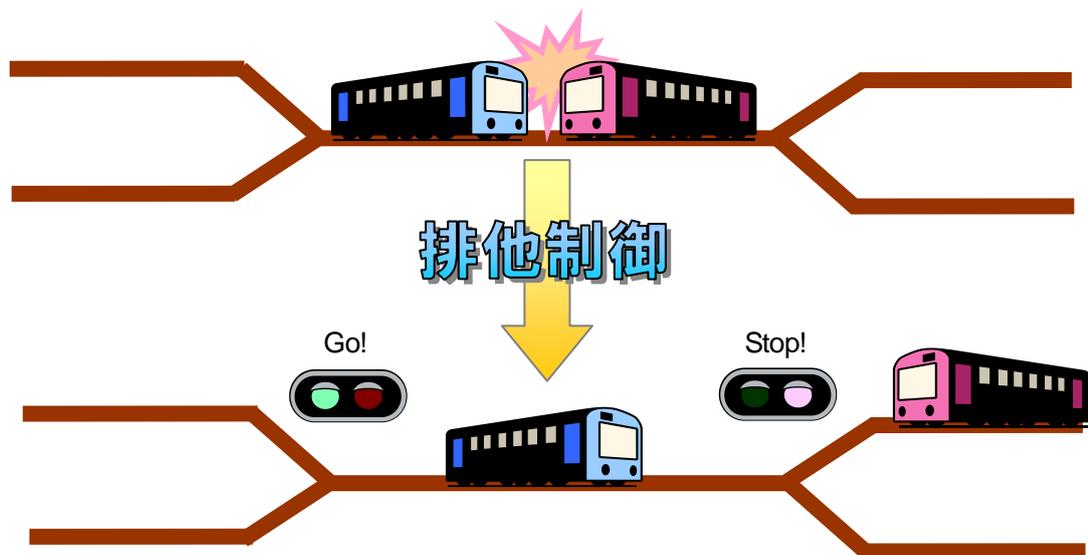



図 6.4-1 排他制御

たとえば、データを管理しているメモリ領域などに複数のタスクがアクセスすると、メモリ領域に意図しない変更を加える可能性があります。そのため、メモリ領域を使用する際のルールを取り決め、かならずメモリを使用するタスクは同時に１つしかない状態を作る必要があります。TOPPERS Automotive Kernelでは、「優先度上限プロトコル」を利用して排他制御を実現しています。

6.4.1 優先度上限プロトコル

リソースを獲得する可能性のあるタスク・ISR の最高優先度（上限優先度）までタスクの優先度を引き上げるスケジューリング方式を優先度上限プロトコルと言います。タスクがリソースを獲得すると優先度を上限優先度に引き上げられ、リソースを解放すると元の優先度に戻ります。つまりリソースを獲得するとリソースを解放するまで、そのリソースを獲得する可能性のあるタスクや ISR が実行されなくなるので、排他制御を実現することができます。

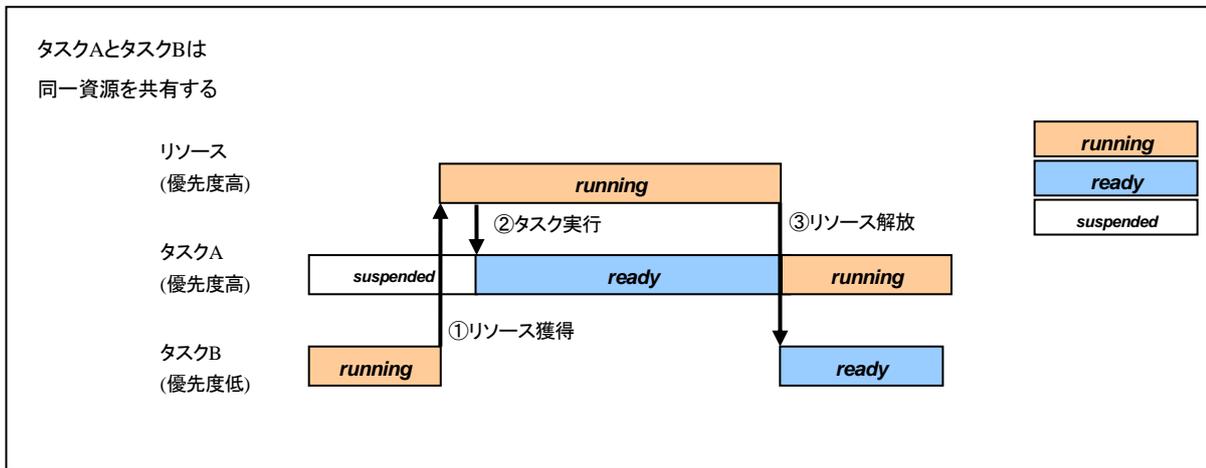



図 6.4-2 優先度プロトコル

図 6.4-2は、タスク A とタスク B が同一資源を共有する場合の動作を示しています。タスク B よりタスク A のほうが高優先度なのでリソースの優先度はタスク A と同じです。

① タスク B がリソースを獲得するとタスク B はタスク A と同一優先度になります。
② タスク B からタスク A を実行します。リソースを獲得しているタスク B は同一優先度のため実行状態を継続します。タスク B は実行可能状態になります。
③ タスク B がリソースを解放するとタスク A はタスク B より優先度が高くなるため実行状態になり、タスク B が実行可能状態になります。

6.4.2 リソースの使用方法

■ リソースの宣言とタスクとの関連付け

リソースも OIL ファイルで定義します。また、排他を行うタスクとリソースを関連付けます。

```
TASK Task1 {                            /* タスク名 */
        PRIORITY = 5;                   /* タスク優先度 */
        ACTIVATION = 1;                 /* 起動要求キューイング数 */
        SCHEDUKE = FULL;                /* スケジューリングポリシー */
        STACKSIZE = 0x200;              /* スタックサイズ */
        RESOURCE = Resource1;           /* 使用するリソース */
};

TASK Task2 {                            /* タスク名 */
        PRIORITY = 3;                   /* タスク優先度 */
        ACTIVATION = 1;                 /* 起動要求キューイング数 */
        SCHEDUKE = FULL;                /* スケジューリングポリシー */
        STACKSIZE = 0x200;              /* スタックサイズ */
        RESOURCE = Resource1;           /* 使用するリソース */
};

RESOURCE Resource1 {                            /* リソース名 */
        RESOURCEPROPERTY    = STANDARD; /* リソース属性 */
};
```

■ リソースの取得と解放

```
GetResource(Resource1);                /* Resource1 の取得 */
            (排他中の処理)
ReleaseResource(Resource1);            /* Resource1 の解放 */
```

6.4.3 リソース機能使用時の注意

リソース獲得状態では以下のことが禁止されています。

- タスクの終了
- スケジューラの呼出し
- イベント待ち
- 割込みハンドラの終了

6.4.4 常に排他制御を意識する

排他制御を行う場合、タスクがこれを意識して使用する必要があります。上記の電車の例でも、信号機は事故を起こさないための手段ではありますが、使う電車が信号機を守らなければ意味がないのと同じです。タスクを作成するときは、排他制御を意識して設計する必要があります。

演習問題

以下の条件を満たす 2 つのタスクを作成し、LCD に文字列を表示するシステムを作成して下さい。

1.タスクはプリエンプティブ のみ使用する
2.2 種類のタスクの優先度は変える
3.タスクは次のタイミングで実行を開始する
・低優先度タスク：500ms 周期
・高優先度タスク：低優先度タスクの開始から 400μs 後
4.LCD には次の文字列を表示する
・低優先度タスク：ABCDEFGHIJKLMNOP
・高優先度タスク：abcdefghijklmnop
5.タスクが実行状態もしくは実行可能状態の間だけ LED を点灯する
・低優先度タスク：LED2 を使用
・高優先度タスク：LED3 を使用

上記システムを実行し、LCD に大文字と小文字が混じって表示されることを確認して下さい。これは LCD という共有資源を同時に利用するために起こる現象です。大文字と小文字が混ざらない場合、高優先度タスクを開始するタイミングをずらして調整して下さい。
次に、上記システムに以下の条件を追加して下さい。

6.文字列を表示する処理の前でリソースを獲得し、後で解放する

上記システムを実行し、LCD に小文字のみ表示されることを確認して下さい。これはリソースを使用したことで排他制御され、LCD という共有資源の同時利用を防いでいることを意味します。大文字の表示は表示時間が非常に短いため目視で確認できませんが、LED2 の点滅で低優先度タスクが実行されていることを確認して下さい。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex6_res」フォルダにセットアップ済みです。

①「H25_toppers_ ex6_res 」フォルダをコピー
②「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

7 MISRA-C

7.1 コーディング規約とは

コーディング規約とは、ソースコードを書く際のルールです。コーディング規約には、可読性を向上させるための**コーディングスタイル**をベースに、**移植性を向上させる**ためのルール、**危険を回避する**ためのルール、より**可読性を向上させる**ためのルールなどがあります。

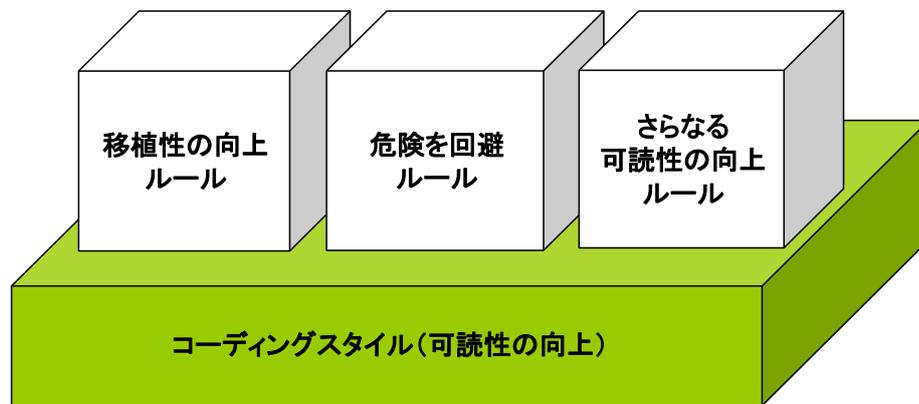



図 7.1-1 コーディング規約イメージ図

コーディング規約は、使用する言語によってそれぞれ存在します。また、プロジェクトごとに独自のコーディング規約を定め、ローカルなルールに従い開発を行う場合があります。本章では、C言語の代表的なコーディング規約であるMISRA-Cについて説明します。

7.1.1 C 言語コーディング規約

C言語コーディング規約には、非常にさまざまなものがあります。以下に主なC言語コーディング規約を示します。

表 7.1-1 C 言語コーディング規約

コーディング規約名	対象・特徴
GNU コーディング規約	GNU ソフトウェア
Indian Hill Style Guide	T&TBell Laboratory で使用されていた、古典的な規約
Linux Kernel Cording Style	Linux カーネル
Google C++ Style Guide	Google Code
MISRA-C	組込みソフトウェア開発向き
IPA/SEC コーディング作法ガイド(ESCR)	MISRA-C、Indian Hill 、GNU などのコーディングルールを参照してまとめたもの
社内独自ルール	社内プロジェクトに合ったようにカスタマイズされている

GNUソフトウェアのコーディング規約であるGNUコーディング規約、Linuxカーネルのコーディング規約であるLinux Kernel Coding Style、Indian Hill Style Guide や C style Guide もあります。最近では、Googleコード向けの、Google C++ Style Guide というものも出てきました。IPA/SECが作成したコーディング作法ガイド（ESCR）は、MISRA-C、Indian Hill、GNUなどのコーディングルールを参照してまとめたコーディング規約です。

また、社内独自で定めたものもコーディング規約と呼べます。社内プロジェクトで必要とされるようにカスタマイズされており、ベストなコーディング規約でしょう。しかし、作成する手間がかかるのが難点です。そのため、既存のコーディング規約をベースに作成されるケースが多いです。今回扱うMISRA-Cもコーディング規約の一種です。

7.1.2 C 言語にコーディング規約が必要な理由

なぜ、C言語にコーディング規約が必要なのでしょうか？

C言語は、記述に対する自由度が高く、使いやすいという利点があります。その反面、コンパイル時のチェックや実行時のチェック機能が弱かったり、C言語自体の規格が動作を明確に定めていない曖昧さがあります。

たとえばポインタを使った操作では、非常に自由度が高く、アクセスしてはいけない領域にアクセスしてしまう危険性や、配列の添え数のチェックがないため、確保されていない領域にアクセスしてしまう危険性があります。

また、C基準となる規格書C90（ISO/IEC9899 : 1990）では、動作をコンパイラメーカーにまかせているため、CPU・コンパイラ依存により、コード再利用時に問題が出る場合があります。つまり、移植性が低いのです。MISRA-Cなどのコーディング規約に従ってコーディングすることにより、危険の少ないコードやコンパイラ依存を極力抑えたコードを書くことができます。

7.2 MISRA-C とは

MISRA（Motor Industry Software Reliability Association）は、ヨーロッパ自動車技術会（MIRA : Motor Industry Research Association）の関連組織で、自動車用ソフトウェアの信頼性に関する組織です。

MISRAは、MISRAドキュメントとして、“Development Guidelines for Vehicle Based Software” を1994年に発表しています。これには、自動車用ソフトウェアの開発ガイドラインが記載されており、**MISRA-SA**と呼ばれています。また、1998年に “Guidelines for The Use Of The C Language In Vehicle Based Software”が発表しています。これは、自動車用C言語コーディング規約が記載されており、**MISRA-C**と呼ばれています。さらに、2004年に “Guidelines for the use of the C language in critical systems”を発表し、こちらの内容は自動車用C言語コーディング規約の第2版であり、自動車に限らず、組込みソフトウェア開発向きのコーディング規約となっていま

す。2つのドキュメントはそれぞれ発表年に従い、**MISRA-C:1998**、**MISRA-C:2004**と区別されています。

MISRA-Cは、2004年版にて改善され、自動車に限らず、**組込みソフトウェア開発向きのコーディングガイドライン**として利用しやすくなりました。欧州、北米、日本をはじめ、世界中で利用されはじめています。自動車業界ではコーディング規約のベースとして広く使われており、MISRA-C対応を必須とする日本の自動車メーカーも存在します。

図 7.2-1 MISRA ドキュメント概略図

7.2.1 MISRA-C の背景

近年自動車の性能は多機能化し、それを実現するために多くのECUが自動車に搭載されることになりました。それに伴い、ECUを制御するためのソフトウェアの開発量が増加しました。組込みソフトウェアは、信頼性、演算速度などの要求が高いため、従来通りの開発の仕方では、このペースの開発量の増加にとても対応することはできません。このような状況下で、欧州の自動車業界では、**開発量の増加と品質維持の対応を目的として、MISRAドキュメントの発表**を行いました。

7.2.2 MISRA-C の効果

MISRA-Cを適用すると、以下2点の効果があげられます。

- 複雑度の低いソースコードになる。
- 重大なバグを減少させることができる。

経路複雑度と1関数あたりの行数は、比例した関係にあります。また、経路複雑度が高くなると、静的経路数は爆発的に増加します。ここでMISRA-Cの違反指摘頻度は、経路複雑度と反比例の関係にあるとデータ報告されています。逆にとらえると、MISRA-Cの違反数が少なくなるようなコードを心がければ、自然と複雑度の低いコードにすることができます。すると、静的経路数を減らすこともでき、その結果テストを容易にできます。また、シンプルなコードは可読性が向上するため、考慮漏れの危険を事前に防ぎやすくなり、移植や機能追加もしやすくなります。

重大バグは、非常に数が少ない上に、想定していないタイミングや条件の組み合わせで発現し、見つけにくいものが多いです。なぜなら、通常のテストで発見できないバグは、非常にまれにしか発生しないものが多いからです。そのため、MISRA-C対応を行い、ルール違反数を減らすことで、**重大なバグを減少させる**ことができます。MISRA-C対応ではすべてのルール違反箇所を再確認するので、確実に**ソフトウェアの品質を向上させる**ことができます。また、MISRA-Cのルール違数を見ることで、重大バグ数を推定することもできます。

7.2.3 MISRA-C の特徴

1. 構成

具体的なプログラミングルールと、品質サブシステムの解説で構成されています。基準となる規格書は、C90（ISO/IEC9899：1990）であり、ルール自体は組込みに特化している訳ではありません。

組込みCプログラムに適した良いリファレンスとしては珍しく公開されたガイドラインです。抽象的な表現で解釈が難しいものがあったり、リスクがわかりにくい点がありますが、よい解説書が出ているため、それでカバ

ーできます。

2.　必要ルールと推奨ルール

　MISRA-Cのルールは「**必要**」と「**推奨**」に分類されています。
必要ルールは、強制的な要求であり、ルールに従えない場合、逸脱手続きが必要です。推奨ルールは、通常従うほうがよい要求であり、ルールに従えない場合やルールに従うことにより可読性や保守性などの低下を招くような場合のみ行えばよいです。
逸脱手続きに関しては、**7.3.4 逸脱手続き方法**にて説明します。

MISRA-Cルールの分類

必要ルール 守れない場合は 逸脱手続きが必須	**推奨ルール** 守れない場合の 逸脱手続きは 必須ではない

図 7.2-2 MISRA-C ルール分類図

3.　MISRA-C：1998とMISRA-C2004の違い

　7.2で述べましたが、MISRA-Cは2種類存在します。1998年に策定された**MISRA-C:1998**と、2004年に策定された**MISRA-C:2004**です。
　2つのルールの違いは、MISRA-C:2004は、MISRA-C:1998のルールを見直し、ルールの追加、廃止、訂正がされています。その結果、車載向けだったものが、広くすべての組込みソフトウェア向けに使えるものになりました。また、あるルールに対応した結果、別のルールがNGになり、必ずどれかのルールに引っかかってしまう、というケースは減りました。
　その他の違いとして、ルール番号の形式が違います。MISRA-C:2004では、21の項目別にルールをまとめ、ルール番号もそれに合わせています。解説書には、MISRA-C:1998とMISRA-C:2004のルール番号の対比表が掲載されているものもあります。
以下が、MISRA-C:1998とMISRA-C:2004の対比です。

表 7.2-1 MISRA-C:1998 と MISRA-C:2004 の比較

	MISRA-C：1998 (Version 1)	MISRA-C：2004 (Version 2)
ルール数	127 項目 必要：93 推奨：34	141 項目 必要：121 推奨:20
ルール番号形式	ルール 1～ルール 127	ルール 1.1～ルール 21.1
開発対象	車載向け	すべての組込みソフトウェア向け
ルール改善点		MISRA-C:1998 のルール 15 項目（ルール 8、18、20、28、44、55、58、79、80、84、104、105、107、113、121）が廃止・訂正 あるルールに対応するとあるルールは NG、というケースが減った。

　どちらのバージョンを用いるかは、製品の開発時期や企業の方針などによりわかれるところであり、

MISRA-C:1998を使用している企業もまだまだ多いです。

7.2.4 MISRA-C ルールの例

ここでは、MISRA-C:2004 の代表的なルールについて、ソースを交え説明を行います。それ以外のルールについては、**12 Appendix B（MISRA-C ルール一覧）**やガイドブック等を参照して下さい。

ルール 5.2 識別子の隠蔽（必要）
外部スコープの識別子が隠蔽されるため、内部スコープの識別子には、外部スコープの識別子と同じ名前を用いてはならない。

以下は、ソースコードによるルールの解説です。

```
void main ( void ) {
    #define status_run( ) ¥
    do { ¥
        unsigned char status; ¥     /* 内部スコープの変数 */     ー①
        status = RUN; ¥
    } while ( 0 )
  …
    unsigned char status = INIT;  /* 外部スコープの変数 */     ー②
    status_run( );
    if (status == RUN) {
       /* 実行時処理 */
    }
```

ルール5.2は、同一有効範囲内で同じ識別子名の再使用を禁止するものです。上記の例では以下の内容が非適合となっています。
①マクロ置き換えにより、内部有効範囲にて外部有効範囲内で使用されている識別子名を再使用しています。
②内部有効範囲での識別子名の再使用により、外部有効範囲の識別子を隠蔽しています。

同じ識別子名が外部有効範囲内、内部有効範囲内に入れ子となっている場合、外部有効範囲の識別子を隠蔽してしまうため、第3者は、開発者がそれを意図して行っているのかが判断できず、保守性を低下させ、プログラムの修正を行う際に混乱が生じます。
回避方法としては、外部有効範囲の変数名にはプロジェクト名を頭に付ける（例: uint16_t prj1_ui16_var ）など、プレフィックスを利用することで回避することができます。

ルール 6.1 char型（必要）
単なるchar型は、文字データの格納及び使用に限って用いなければならない。

以下は、ソースコードによるルールの解説です。

```
void func2(void) {
        char  val1, val2;

        val1 = -3 ;                 －①
        val2 =  1 ;

        if( val1 + val2 < 0 ) {     －②
              ・・・
        }
 }
```

ルール6.1は、より安全なソースコードを記述するために、処理系によって扱いの異なる可能性のある、単なるchar型の仕様を文字データに制限するルールです。

char型変数が符号付で扱われる場合には、マイナス値（-128 ～ -1）はそのまま扱われますが、char型変数が符号なしで扱われる場合、以下のようにマイナス値は整数値（128 ～ 255）に繰り上げて扱われてしまいます。

① 符号付きの場合　　　　　　符号なしの場合
　val1 = -3;　　　　　　　　　val1 = 253;
　val2 = 1;　　　　　　　　　 val2 = 1;

そのため、②の計算式の結果は符合ありで扱われるか、符号なしで扱われるかによって、以下のように結果が異なることになります。

② 符号付きの場合　　　　　　符号なしの場合
　-3 + 1 = -2　　　　　　　　 253 + 1 = 254

char型が符号付きで扱われるか、符号なしで扱われるかは処理系に依存します。単なるchar型の符号を認識せずにchar型の数値データを使用すると、符号の扱いの異なる処理系に移植した際に、プログラムが予期しない動作をしてしまう可能性があり、移植の際には、すべてのソースコードの見直しが必要となります。また、signed char型、unsigned char型は数値データのみの使用制限とされます。(ルール6.2)

ルール 7.1 （0以外）の8進定数（必要）
(0以外の)8進定数及び8進拡張表記は、用いてはならない。

以下は、ソースコードによるルールの解説です。

```
VAR1 = 100    10進定数として判断          適合  －①


VAR2 = 070
VAR3 = 065     0で始まる整数定数は
VAR4 = 055     8進定数として判断          非適合  －②


VAR5 = ¥109
VAR6 = ¥100   8進拡張表記による判断       非適合  －③
```

ルール7.1は、8進定数及び、8進拡張表記の使用に起因する問題を回避するために、8進定数及び8進拡張表記の使用を禁止しています。

①10進数の100を設定しています。→適合

②0で始まる整数定数は8進定数として判断されます。→非適合
上記理由のため、VAR2～VAR4の値は以下のようになります。
VAR2 = 10進数の56
VAR3 = 10進数の53
VAR4 = 10進数の45
③8進拡張表記による判断がされます。→非適合
上記理由のため、VAR5～VAR6の値は以下のようになります。
VAR5 = ¥10と9の2文字に解釈されます
VAR6 = 10進数の64

カラム合わせのためなど、先頭に0を不用意に記述すると、その定数は8進定数として扱われてしまい、意図した定数値とならない可能性があります。8進拡張表記を用いた際、8進数字以外の数字を記述すると、2バイト以上の文字定数となる場合があります。

ルール 9.1 自動変数（必要）
すべての自動変数は、用いる前に値を代入しなければならない。
自動変数は使用する前に値を代入し、自動変数が不定の値にならないようにすべきである。

以下は、ソースコードによるルールの解説です。

```
int count ;
int max_count = 10 ;                    適合ー①
int sum ;                               非適合ー②
int val1 = 1;                           適合ー③
for(count = 0 ; count < max_count ; count++) {
      sum += (val1*count);
}
```

①、③は使用前に初期化がされていますが、②は使用前に初期化されていません。

ルール 14.8 繰返し文の本体（必要）
switch,while,do…while,for文の本体を構成する文は、複合文でなければならない。

ルール14.8は、可読性を向上し、間違いを防ぐためのルールです。複合文とは波括弧で囲まれた文のことです。while、do…while、及びfor文の本体が単一の場合、文法上は波括弧で囲む必要はありませんが、波括弧で囲むことにより、可読性が向上し、保守性が向上します（例１参照）。

以下は、ソースコードによるルールの適合・非適合の例です。

```
int count;
int max_count;
int sum;
int val1;
for(count = 0 ; count < max_count ; count++) {            適合ー①
      sum += (val1*count);
}
```

①for文後に複合文が続いているため適合です。

```
int blood_flag;
while(blood_flag)                        非適合－①
    printf(“A”);
```

①本体が波括弧で囲まれていないため不適合です。

ルール 14.9 “if (式) ”の後（必要）

“if (式) ” 構造の後には、複合文を続けなければならない。elseキーワードの後には、複合文または他のif文を続けなければならない。

ルール14.9は、可読性を向上し、間違いを防ぐためのルールです。If が「真」及び「偽」のときに実行される文が単一の場合、文法上は波括弧で囲む必要はありませんが、波括弧で囲むことにより、可読性が向上し、保守性が向上します。

以下は、ソースコードによるルールの解説です。

```
int blood_type;
int parents_blood_type;
if(1 == blood_type)                       非適合－①
    if(3 == parents_blood_type){
        printf(“blood_type is AO.”);
    }
else if (2 == blood_type) {
    printf(“blood_type is B.”);
}
else if (3 == blood_type) {
    printf(“blood_type is O.”);
}
else if (4 == blood_type) {
                                          適合－②
}
else
    printf(“blood_type is AB.”);          非適合－③
```

①波括弧がないため、elseif文がどのif文の制御式によって選択されているかわかりにくいです。
②中身が空でも複合文であれば適合です。
④ elseに続く文が複合文ではありません。この文以降はif文の範囲外にあるため、制御式の結果に関係なく実行されます。

7.3 MISRA-C 実施プロセス

ここでは、MISRA-Cを実施する際のプロセスについて説明します。**図 7.3-1**はMISRA-Cを実施する際の作業の流れを示しています。

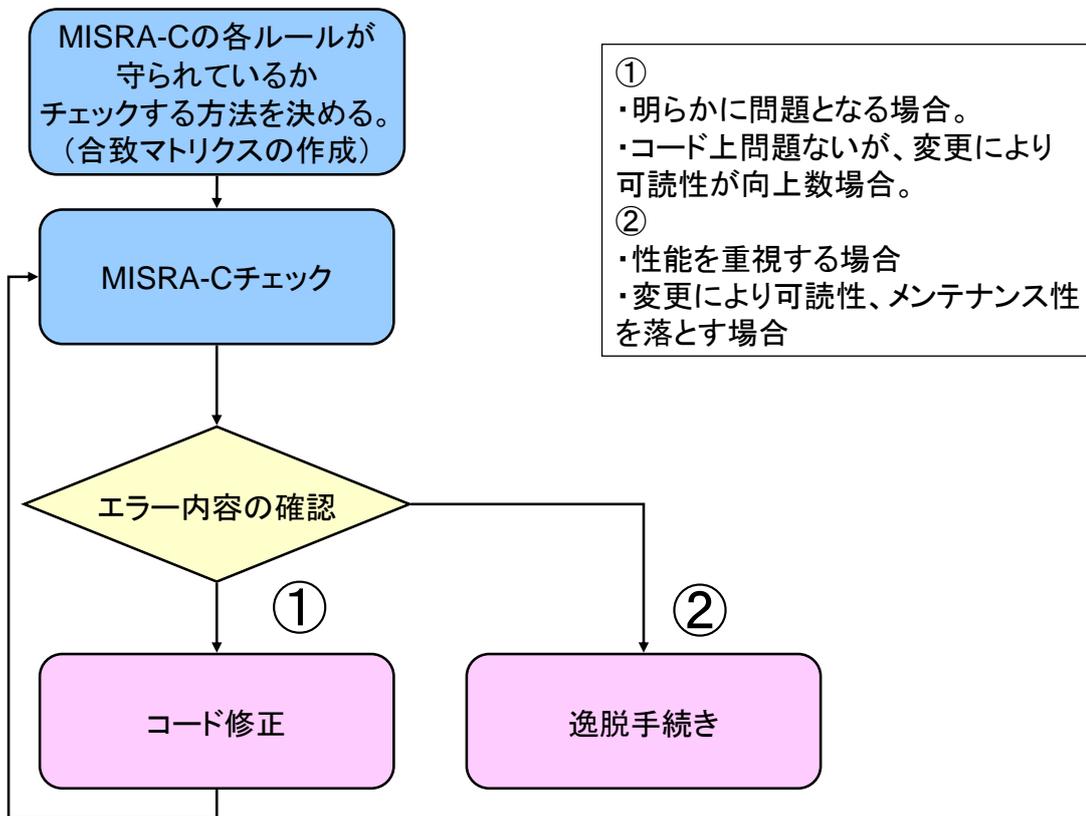



図 7.3-1 MISRA-C 実施のプロセス

最初に、MISRA-Cのルールが守られているかどうかをチェックする方法を決めていきます。チェックする方法は、各ルールごとに決め、ルールとそのチェック方法をまとめたものを合致マトリクスといいます。合致マトリクスの作成は、プロジェクト初期段階でプロジェクトの方針を立ててから行います。以降、基本的に変更することはありません。合致マトリクスの詳細に関しては**7.3.1 合致マトリクス** を参照して下さい。

次に、MISRA-C対応の実作業です。まず、MISRA-Cチェックを行います。合致マトリクスに記載している方法を用い、各ルールが守られているかチェックします。方法としては、コードレビューにより目視でチェックすることも可能ですが、静的解析ツールを利用すると便利でしょう。コーディングを終え、コンパイルを通したあとに、必ずチェックを行うとよいです。MISRA-Cチェック方法の詳細に関しては**7.3.2 MISRA-Cチェック方法**を参照して下さい。

続いて、チェック結果の確認を行います。静的解析ツールでチェックすると、指摘箇所（エラー）の一覧が出てきます。それらのエラーを１つずつ、合致マトリクスを参照して確認していきます。MISRA-Cでは、すべてのエラーを修正する必要はありません。すべてのルールを無理に守ると、逆に品質（可読性・パフォーマンス）を低下させるおそれがあります。明らかに問題となる場合や、コードとしては問題がなくても、変更により可読性・メンテナンス性が向上する場合は、積極的にコードを修正すべきです。コード修正した後は、再度MISRA-Cチェックをする必要があります（図の経路①を参照）。エラーの対処方法については「**7.3.3 エラー対応方法**」を参照して下さい。

MISRA-Cチェックで指摘されたものでも、性能を重視するため、可読性・メンテナンス性を上げるためにわざとそのような実装をしている場合、ルールを守れない理由を文書化し管理することでMISRA-Cを遵守することができます。その工程を**逸脱手続き**といいます。（図の経路②を参照）逸脱手続き方法については「**7.3.4 逸脱手続き方法**」を参照して下さい。

7.3.1 合致マトリクス

前述しましたが、MISRA-Cでは、すべてのルールについて、適合を確認するための手段を明記した表を作成することが必要です。この表を**合致マトリクス**といいます。MISRA-Cのチェック計画時には、合致マトリクスを作成します。**表 7.3-1**は合致マトリクスの例です。

表 7.3-1 合致マトリクスの例

逸脱するルールについても、あらかじめ定めておく。

ルール番号	仮想コンパイラ A	仮想 MISRA-C チェッカーB	人手によるレビュー	備考
1.1	警告 123			
1.2		エラー45		
1.3		警告 789		制限除外
1.4			レビュー実施	ツールチェック不能
・ ・				
21.1	エラー99			

ツールチェックできない場合、コードレビューでチェックする。

合致マトリクスでは、MISRA-Cの各ルールのエラーをどのように検出するか、検出後の対応（制限除外の有無）をあらかじめ決めておきます。そうすることで、チェック漏れを防ぐことができます。エラーの検出方法は、MISRA-C対応のルールチェッカー（静的解析ツール）だけではありません。コンパイラがエラーや警告メッセージを出すものもあります。また、チェッカーも万能ではありませんので、エラー検出できないルールも存在します。その場合は、ソースコードレビューで確認できます。

制限除外、すなわち逸脱手続きを実施するルールについても、プロジェクト方針としてあらかじめ決めておきます。

また、例ではMISRA-Cのルールしか記載していませんが、プロジェクト独自のローカルなルールを合致マトリクスに追加することも可能です。

7.3.2 MISRA-C チェック方法

MISRA-Cのチェックには、MISRA-C対応のルールチェッカーを用います。コンパイラで確認できるものもあります。また、どちらも使用できない場合はソースコードレビューでチェックを行います。

下表にMISRA-C対応のルールチェッカーの一例を示します。

表 7.3-2 MISRA ルールチェッカーの一例

ツールの種類	ツール名称	ツールベンダー
MISRA-C ルールチェッカー	SQMint	ルネサステクノロジ
コードチェックツール	QA MISRA	英 Programming Research 社
	PolySpace	The Mathworks
	Review –C	NEC 通信システム
	Telelogic Logiscope	スウェーデン Telelogic AB
	PGRelief	富士通ソフトウェアテクノロジーズ
CASE ツール	CasePlayer2	GAIO
	Development Assistant for C	RistanCASE 社
自動ユニットテストツール	C++ test	米 Parasoft Corp.

MISRA-C対応ルールチェッカーには、非常に多くのものがあります。純粋なルールチェッカーは、ルネサステクノロジ社製「SQMlint」です。他にも多数のチェックツールがありますが、ほとんどがベースとなるツールに、MISRA-Cのチェッカーが含まれているものです。また、設計・開発を支援するCASEツールにも、コーディング工程で使用するMISRA-Cチェッカーを搭載したものもあります。他に、MISRA-Cチェックのような静的解析をテストの一種と位置付け、搭載しているテストツールもあります。

7.3.3 エラー対応方法

MISRA-Cチェックで指摘された箇所の確認を行います。

明らかに問題となる場合、コードとしては問題がなくても、変更により可読性・メンテナンス性が向上する場合は修正を行います。

また、性能を重視する場合、変更により可読性・メンテナンス性を落とす場合は逸脱手続きを行います。

7.3.4 逸脱手続き方法

MISRA-Cチェックで指摘されたものでも、そのルールを適用できない場合、適用がふさわしくない場合はルールを逸脱する必要性と安全性の確保を文書化し管理する必要があります。そのような作業を**逸脱手続き**といい、作成する文書は**逸脱手続書**と呼ばれています。

逸脱手続書には、主に下表の5項目を記載します。

表 7.3-3 逸脱手続書の記載項目

記載項目	内容
逸脱の発生場所	製品名称、製品バージョン、モジュール名称、ファイル名称、関数名称、オブジェクト名称（構造体・変数のこと）
逸脱するルール	ルール番号、ルール分類（必要・推奨）、ルール内での逸脱箇所
逸脱による潜在的問題	ルール逸脱で予想される問題
逸脱の必要性	逸脱が必要な理由、利得
逸脱の安全性 （潜在的問題の回避証明）	潜在的問題を回避する方策、潜在的問題が回避されていることの確認手段（安全性の確認と結果）

逸脱の発生場所として、製品名称、製品バージョン、モジュール名称、ファイル名称、関数名称、オブジェクト名称（構造体・変数のこと）を明確にします。以降の内容が同じ場合、複数個所を記載しても構いません。

逸脱するルールとして、ルール番号、ルール分類（必要・推奨）、ルール内での逸脱箇所を明確にします。

逸脱による潜在的問題には、ルール逸脱で予想される問題を記載します。これは、MISRA-C のガイドブックにて解説されていますので、参考にするとよいでしょう。

逸脱の必要性には、逸脱が必要な理由を説明します。基本的には、プロジェクトの方針がどうなっているかを説明することになります。

逸脱の安全性では、逸脱しても問題ないことを証明します。逸脱するということは、前述の潜在的問題が発生しますので、それをどう回避するか、方策を示します。また、潜在的問題が回避されていることをどう確認したかを示します。たとえば、逸脱している箇所のコードレビューすることで回避、等があります。

演習問題

次の演習 1、演習 2 には MISRA-C に違反したプログラムの記述がされています。各々違反しているルールと、どのように直すべきかを述べて下さい。

演習 1

```
unsigned int status;

#define STATE1	((unsigned int)(0U))
#define STATE2	((unsigned int)(1U))

void func(void)
{
	unsigned int status;
	unsigned char input;

	/* func2は外部に宣言されている関数とする */
	if(func2(input) == 002)
	{
		status = STATE1;
	}
	else
		status = STATE2;

	return;
}
```

演習 2

```
#define ARRAY_SIZE	(100)

void func(void)
{
	char i;
	unsigned int array[ARRAY_SIZE];

	for(i = 0 ; i < ARRAY_SIZE ; i++)
		array[i] = i;

	return;
}
```

8 デバイスドライバ

8.1 デバイスドライバとは

パソコンを使用している人ならば、一度はデバイスドライバという言葉を耳にしたことがあるでしょう。

まず、デバイスドライバの「デバイス（device）」とは、一般的にプリンタなどの周辺装置という意味を持ちます。また、リアルタイム OS を使用した組込みシステムでは、LAN（Local Area Network）コントローラなどの電子部品の意味をもちます。この意味から推測してわかるようにデバイスドライバとは、「**周辺装置もしくは電子部品を制御するソフトウェア（プログラム）**」を意味します。

8.1.1 デバイスドライバの役割

組込みシステムにおけるデバイスドライバは、どのような役割を果たしているのでしょうか。デバイスドライバは、アプリケーションもしくは OS とデバイス間の情報の橋渡しをしています。

たとえば、携帯電話を例にして説明します（**図 8.1-1**）。

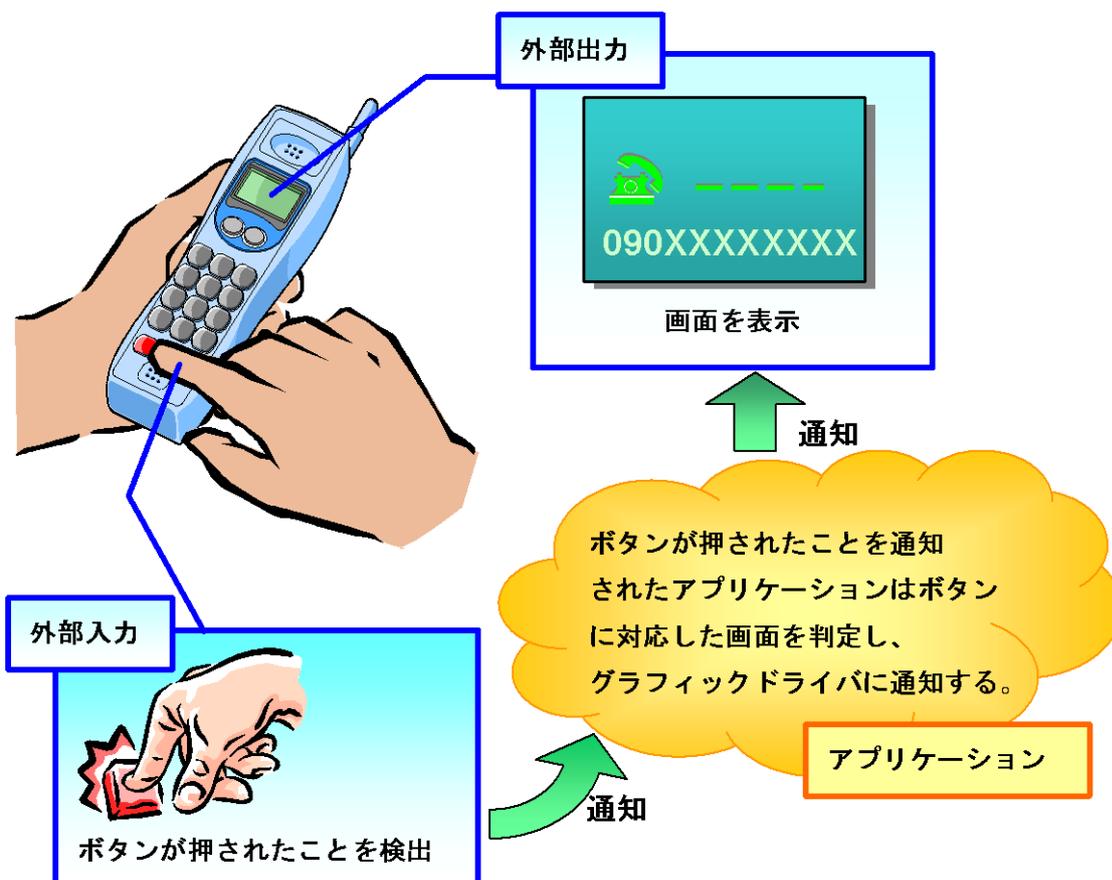



図 8.1-1 デバイスドライバの役割

あなたが、携帯電話を使用して友達に電話をかけるとします。電話をかけるためには携帯電話のボタンを押します。すると、携帯電話の内部ではボタンが押されたという情報が検出されます。その情報は外部入力としてデバイスドライバに通知されます。デバイスドライバはボタンが押されたことをアプリケーションに通知します。アプリケーションは、押されたボタンを判定し、ディスプレイ上にデバイスドライバを通じて外部出力として表示します。このようにデバイスドライバは外部入力及び出力をするための処理を行っているのです。

8.1.2 デバイスドライバの構造

デバイスドライバはどのような構造で作成されるのが良いのでしょうか。組込みシステムでは一般的にデバイスにアクセスするアプリケーションを 3 つの階層（アプリケーション、アプリケーション I/F、デバイス I/F）に分けて考えます（**図 8.1-2**）。

アプリケーション I/F は汎用性を高めるために、抽象化された I/O ライブラリを作成します。デバイス制御部は実際の物理デバイスとアクセスする部位であり、デバイスに対してプログラムから入出力を行い、デバイスの制御を行います。
アプリケーション I/F 部とデバイス I/F 部と分けて作成することにより実装するデバイスが異なっても変更する部分が少なくてすみます。

また、デバイスを階層化することにより、アプリケーションからはデバイスの処理を意識せず API を呼ぶだけで使用することができるようになります。
図 8.1-3 に、デバイスドライバを含めたアプリケーションとハードウェアの制御について示します。

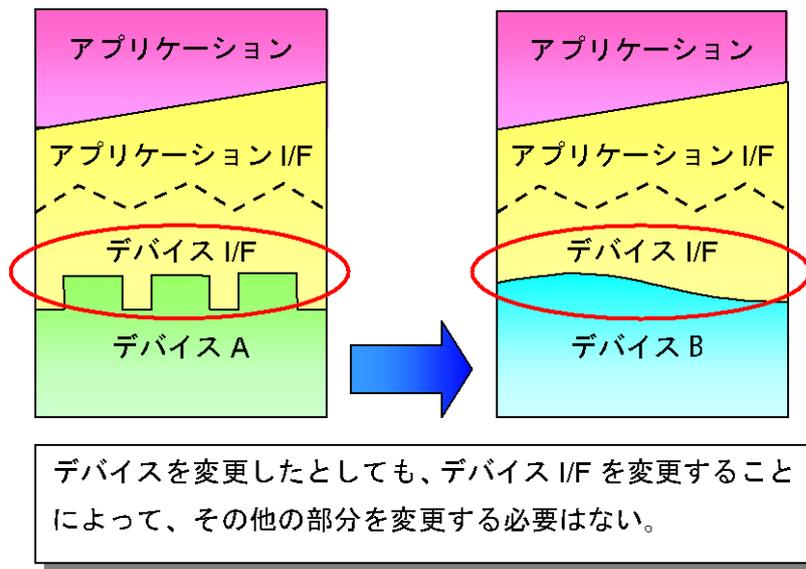



図 8.1-2 デバイスドライバの構造

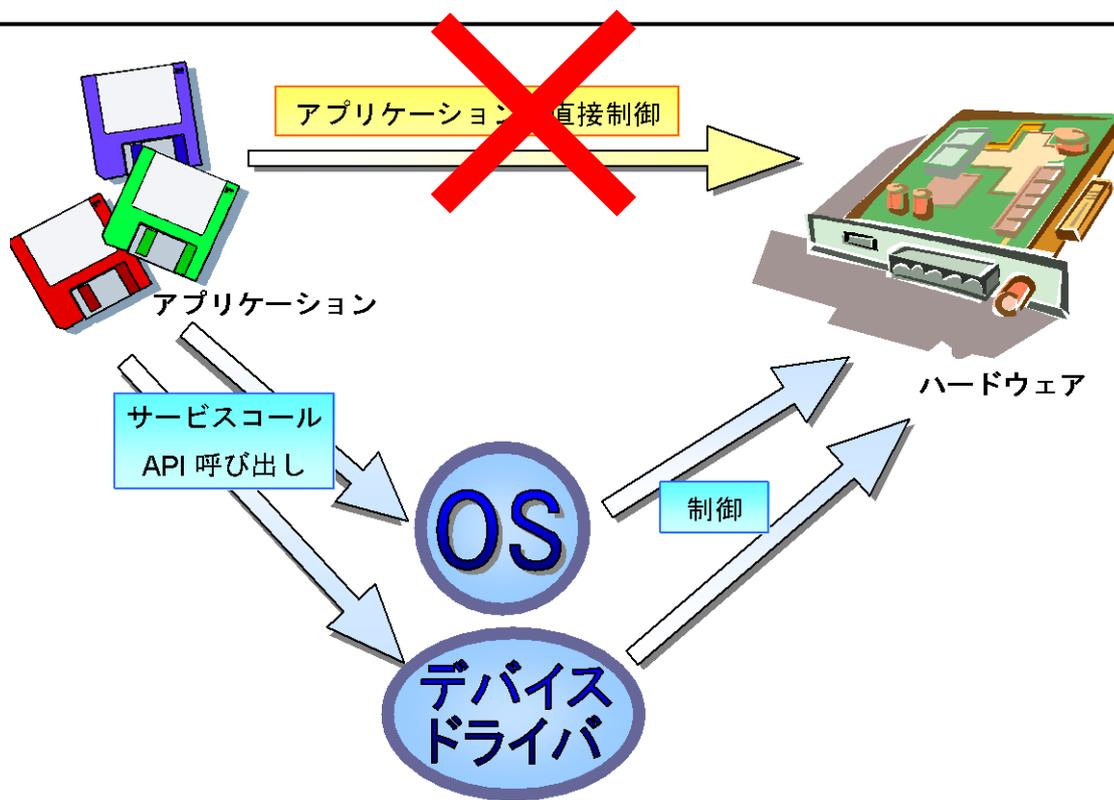



図 8.1-3 ハードウェアの抽象化による制御

8.2 デバイスドライバを使う

本節では、シリアル通信のデバイスドライバを使用したプログラムを作成してみましょう。

8.2.1 提供デバイスドライバについて

M32C 基板用に実装したデバイスドライバの基本構造について説明します。

本デバイスドライバは、**図 8.2-1** のようにデバイスドライバ、アプリケーションインタフェース（アプリケーション I/F）、アプリケーションというように抽象化された構造になっています。そのため、今回作成するアプリケーションからはアプリケーション I/F として提供されている API を呼び出すことでデバイスドライバを使用することができます。

たとえば、serial.c 内のシリアル API を呼び出せば、M32C の仕様を気にすることなく、M32C に依存した hw_serial.c の機能を使用することができます。

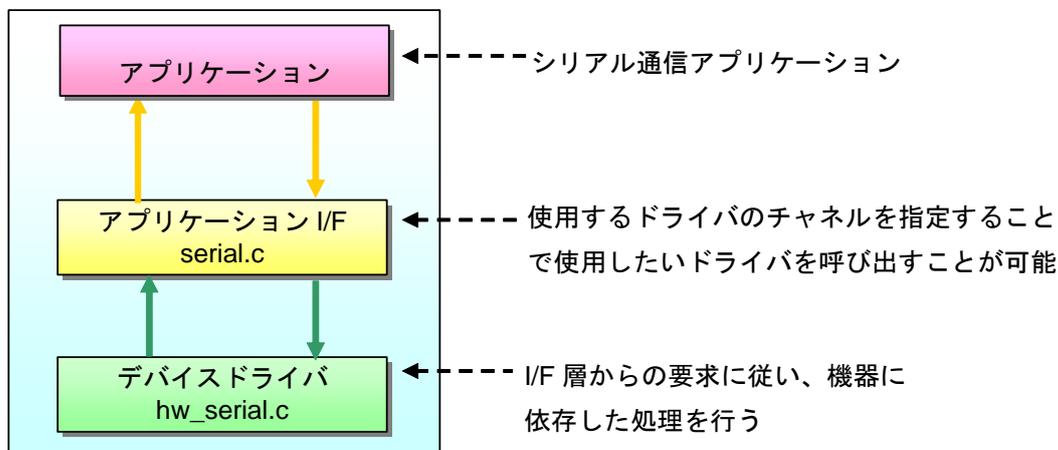



図 8.2-1 シリアルドライバの階層構造例

実際にアプリケーションで使用する前に、初期化と入出力の API について説明します。API を使用した場合のおおまかな処理の流れは **図 8.2-2** のようになります。

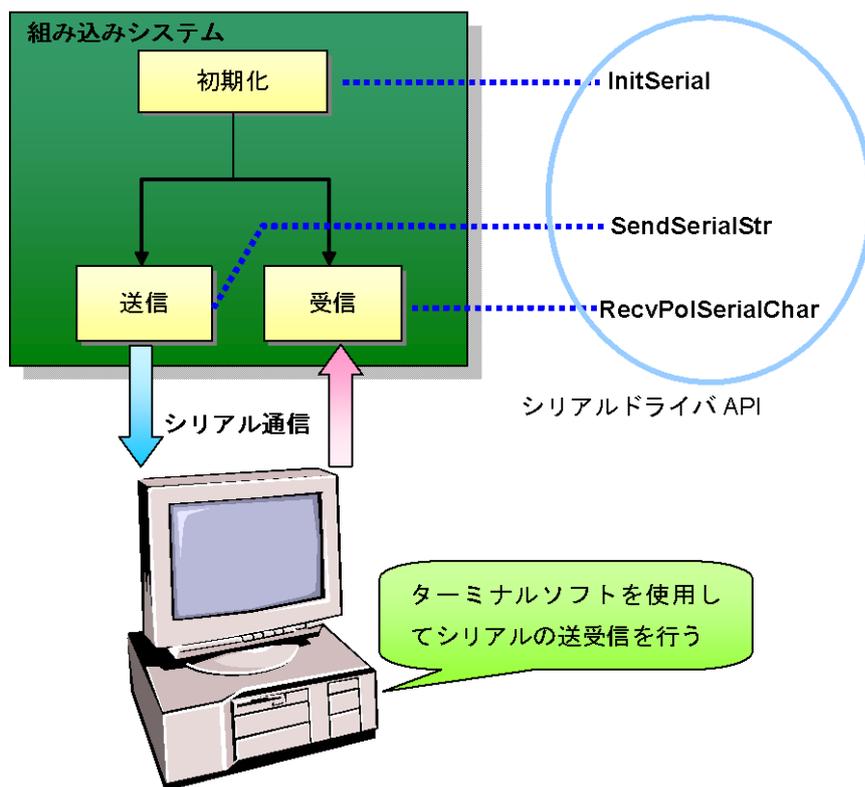



図 8.2-2 API を使用した処理の流れ

以下にシリアルドライバの API の仕様を示します。

8.2.1.1　シリアルドライバ API

シリアルドライバ用の API を用いることで、シリアル通信を行うことができます。**表 8.2-1～表 8.2-6**にシリアルドライバ API の仕様を示します。また、**図 8.2-3～図 8.2-10**に API 呼出し記述例とフローチャートを示します。
使用する前に必ず初期化（InitSerial）を一度呼出して下さい。

表 8.2-1 シリアル API 一覧

API 名	関数名	内容
シリアル初期化 API	InitSerial	シリアルポートの初期化をします。ボーレート、転送データフォーマット等の接続設定は静的に決まります。
シリアル 1 文字入力 API	RecvPolSerialChar	シリアル経由で受信したキャラクタ文字を*character に返します。受信したキャラクタが無ければ、¥0(ヌル文字)を返します。
シリアル文字列出力 API (タスクコンテキスト用)	SendSerialStr	str で指定した文字列をシリアル経由で送信します。タスクコンテキストでのみ使用します。
シリアル文字列出力 API (割込み中用)	SendIntSerialStr	str で指定した文字列をシリアル経由で送信します。割込み中でのみ使用します。
シリアル終了 API	TermSerial	シリアルポートの使用を終了します。

表 8.2-2 シリアル初期化 API　InitSerial

項目		内容
形式		void InitSerial(void);
関数仕様		シリアルポートの初期化をします。ボーレート、転送データフォーマット等の接続設定は静的に決まります。
戻り値	型	Void

	コメント		戻り値なし
引数	1	型	void
		名称	
		コメント	無し

```
InitSerial();
```

図 8.2-3 シリアル初期化記述例

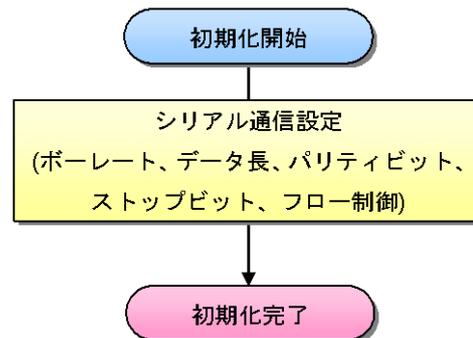



図 8.2-4 シリアル初期化フローチャート

表 8.2-3 シリアル１文字入力 API　RecvPolSerialChar

項目			内容
形式			void RecvPolSerialChar (UINT8 * character);
関数仕様			シリアル経由で受信したキャラクタ文字を*character に返します。 受信したキャラクタが無ければ、¥0(ヌル文字)を返します。
戻り値	型		void
	コメント		無し
引数	1	型	UINT8 *
		名称	character
		コメント	受信文字格納先

```
RecvPolSerialChar( &c );
```

図 8.2-5 シリアル１文字入力 API 記述例

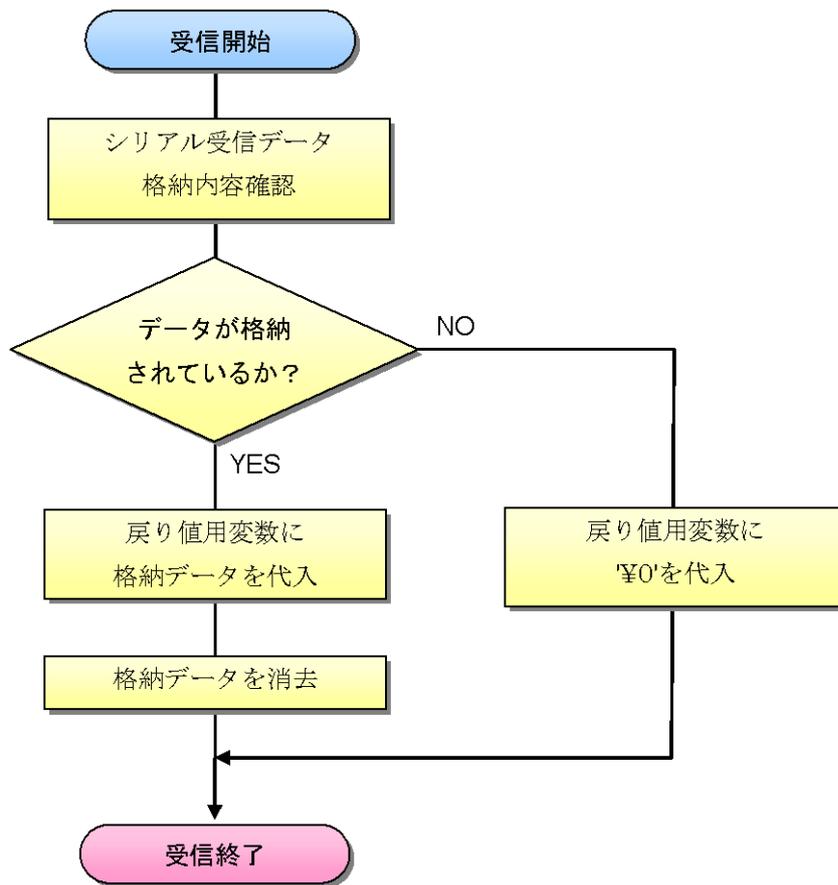



図 8.2-6 データ受信時のフローチャート

表 8.2-4 シリアル文字列出力 API(タスクコンテキスト用)　SendSerialStr

項目			内容
形式			void SendSerialStr (const UINT8 * str);
関数仕様			str で指定した文字列をシリアル経由で送信します。 タスクコンテキストでのみ使用します。
戻り値	型		void
	コメント		無し
引数	1	型	const UINT8 *
		名称	str
		コメント	出力文字列

表 8.2-5 シリアル文字列出力 API(割込み中用)　SendIntSerialStr

項目			内容
形式			void SendIntSerialStr (const UINT8 * str);
関数仕様			str で指定した文字列をシリアル経由で送信します。 割込み中でのみ使用します。
戻り値	型		void
	コメント		無し
引数	1	型	const UINT8 *
		名称	str
		コメント	出力文字列

```
SendSerialStr("Hello¥r¥n");

SendIntSerialStr("Hello¥r¥n");
```

図 8.2-7 シリアル文字列出力 API 記述例

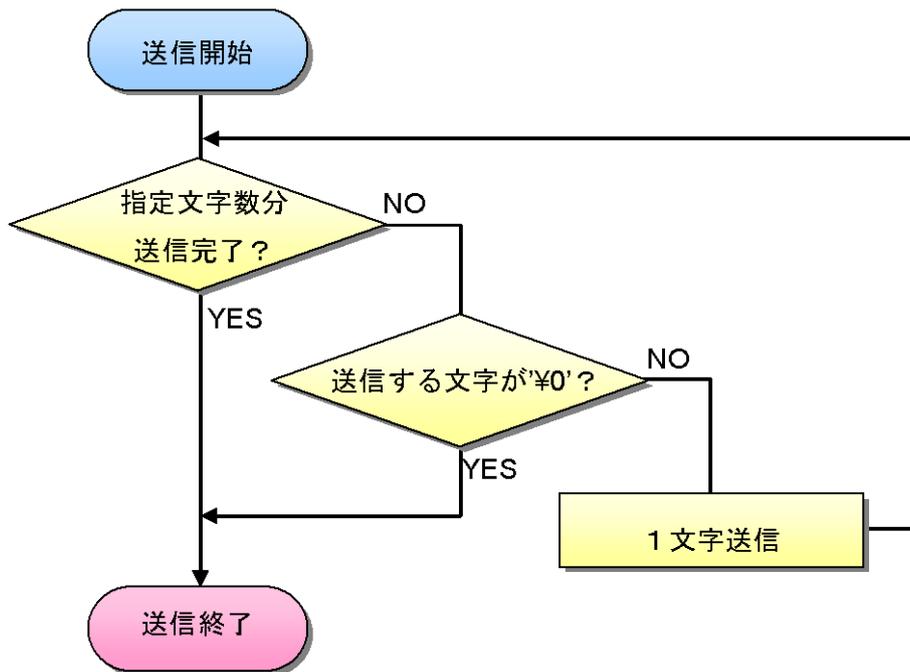



図 8.2-8 シリアル文字列出力のフローチャート

表 8.2-6 シリアル終了 API　TermSerial

項目			内容
形式			void TermSerial (void);
関数仕様			シリアルポートの使用を終了します。
戻り値	型		void
	コメント		無し
引数	1	型	void
		名称	
		コメント	無し

```
TermSerial();
```

図 8.2-9 シリアル終了 API 記述例

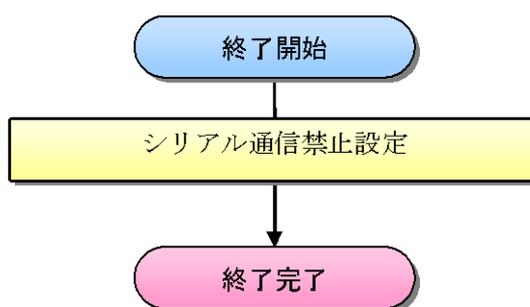



図 8.2-10 シリアル終了フローチャート

8.2.1.2　　LCD ドライバ API

LCD ドライバ用の API を用いることで、LCD 表示を行うことができます。**表 8.2-8** LCD 初期化 API　LCDInit ～**表 8.2-16**に LCD ドライバ API の仕様を示します。また、**図 8.2-11**～**図 8.2-26**に API 呼出し記述例とフローチャートを示します。

使用する前に必ず初期化(LCDInit)を一度呼出して下さい。

表 8.2-7 LCD ドライバ API 一覧

API 名	関数名	内容
LCD 初期化 API	LCDInit	LCD ドライバを初期化し、LCD を使用する準備をします。
LCD 終了 API	LCDTerm	LCD ドライバの使用を終了します。
LCD 表示制御 API	LCDCtlDisplay	LCD の様々な表示制御機能を実行します。
LCD カーソル設定 API	LCDSetCsrPos	LCD のカーソル位置を設定します。
LCD データ 1 文字 読み込み API	LCDRead	LCD のデータを 1 文字読み込みます。 現在位置のカーソルの文字を読み込んだ後、カーソルを 1 つ次へ移動させます。
LCD データ 1 文字 書込み API	LCDWrite	LCD へデータを 1 文字書込みます。 現在位置のカーソルの文字を書き込んだ後、カーソルを 1 つ次へ移動させます。
LCD データ文字列 読み込み API	LCDReadLine	LCD のデータを 1 行読み込みます。 指定行の文字列を読み込み後、カーソルは元の位置に戻ります。
LCD データ文字列 書込み API	LCDWriteLine	LCD へデータを 1 行書込みます。 指定行で文字列を書き込んだ後、カーソルは元の位置に戻ります。

表 8.2-8 LCD 初期化 API　LCDInit

項目			内容
形式			LCD_RET LCDInit(void);
関数仕様			LCD ドライバを初期化し、LCD を使用する準備をします。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　　：正常終了 LCD_E_TMOUT：タイムアウトエラー
引数	1	型	void
		名称	
		コメント	無し

```
LCDInit();
```

図 8.2-11 LCD 初期化 API 記述例

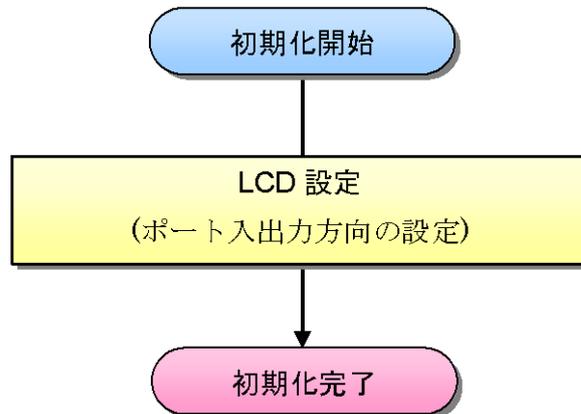



図 8.2-12 LCD 初期化のフローチャート

表 8.2-9 LCD 終了 API　LCDTerm

項目			内容
形式			LCD_RET LCDTerm(void);
関数仕様			LCD ドライバの使用をします。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　: 正常終了 LCD_E_STS : 状態異常
引数	1	型	void
		名称	
		コメント	無し

```
LCDTerm();
```

図 8.2-13 LCD 終了 API 記述例

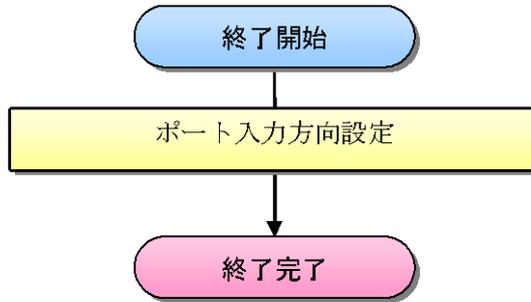



図 8.2-14 LCD 初期化のフローチャート

表 8.2-10 LCD 表示制御 API　LCDCtlDisplay

項目			内容
形式			LCD_RET LCDCtlDisplay (LCD_CH_NO ch_no, LCD_CTL_CODE ctl_code);
関数仕様			LCD のさまざまな表示制御機能を実行します。 機能一覧は**表 8.2-11**を参照して下さい。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　　: 正常終了 LCD_E_TMOUT : タイムアウトエラー LCD_E_PRM　: 引数異常 LCD_E_STS　: 状態異常
引数	1	型	LCD_CH_NO
		名称	ch_no
		コメント	チャネル番号
	2	型	LCD_CTL_CODE
		名称	ctl_code
		コメント	制御コード(**表 8.2-11**参照)

表 8.2-11 LCD 表示制御用コード一覧

制御コード	機能
LCD_CTL_CLRDISPLAY	ディスプレイクリア
LCD_CTL_CSROFF	カーソル表示を消去
LCD_CTL_CSRON	カーソルを表示
LCD_CTL_CSRBLINK	カーソルを点滅表示
LCD_CTL_CSRHOME	カーソルをホーム(1 行 1 文字)へ移動
LCD_CTL_CSRCRLF	カーソルを次の行の先頭へ移動
LCD_CTL_CSRCR	カーソルを同一行の先頭へ移動
LCD_CTL_CSRLF	カーソルを次の行の同一桁へ移動
LCD_CTL_CSRRIGHT	カーソルを右へ移動
LCD_CTL_CSRLEFT	カーソルを左へ移動
LCD_CTL_DSPRIGHT	表示を右へスクロール
LCD_CTL_DSPLEFT	表示を左へスクロール

```
LCDCtlDisplay ( 0, LCD_CTL_CLRDISPLAY );
```

図 8.2-15 LCD 表示制御 API 記述例

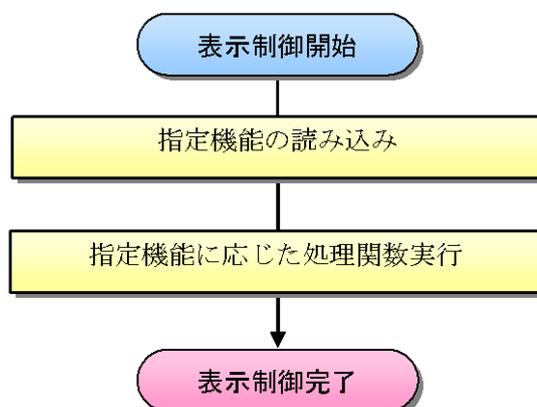



図 8.2-16 LCD 表示制御のフローチャート

表 8.2-12 LCD カーソル設定 API　LCDSetCsrPos

項目			内容
形式			LCD_RET LCDSetCsrPos (LCD_CH_NO ch_no, UINT8 csr_line, UINT8 csr_digit);
関数仕様			LCD のカーソル位置を設定します。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　　: 正常終了 LCD_E_TMOUT : タイムアウトエラー LCD_E_PRM　: 引数異常 LCD_E_STS　: 状態異常
引数	1	型	LCD_CH_NO
		名称	ch_no
		コメント	チャネル番号
	2	型	UINT8
		名称	csr_line
		コメント	行指定(1 or 2 を指定)
	3	型	UINT8
		名称	csr_digit
		コメント	桁指定(1 ～ 16 を指定)

```
LCDSetCsrPos( 2, 2, 2 );
```

図 8.2-17 LCD カーソル設定 API 記述例

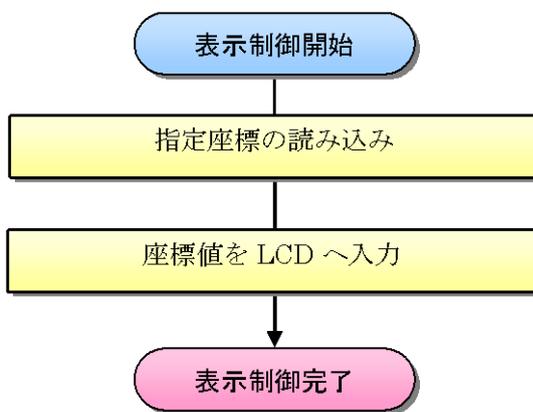



図 8.2-18 LCD カーソル設定のフローチャート

表 8.2-13 LCD データ 1 文字読み込み API　LCDRead

項目			内容
形式			LCD_RET LCDRead (LCD_CH_NO ch_no, LCD_CHARACTER *p_data);
関数仕様			LCD のデータを 1 文字読み込みます。 現在位置のカーソルの文字を読み込んだ後、カーソルを 1 つ次へ移動させます。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　　: 正常終了 LCD_E_TMOUT : タイムアウトエラー LCD_E_PRM　: 引数異常 LCD_E_STS　: 状態異常
引数	1	型	LCD_CH_NO
		名称	ch_no
		コメント	チャネル番号
	2	型	LCD_CHARACTER *
		名称	p_data
		コメント	文字格納アドレス

```
LCDRead( 0, &c );
```

図 8.2-19 LCD データ 1 文字読み込み API 記述例

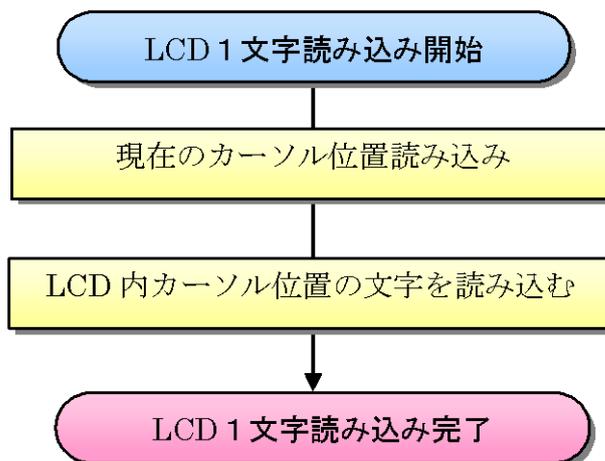



図 8.2-20 LCD データ 1 文字読み込みのフローチャート

表 8.2-14 LCD データ 1 文字書込み API　LCDWrite

項目			内容
形式			LCD_RET LCDWrite (LCD_CH_NO ch_no, LCD_CHARACTER data);
関数仕様			LCD へデータを 1 文字書込みます。 現在位置のカーソルの文字を書き込んだ後、カーソルを 1 つ次へ移動させます。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　: 正常終了 LCD_E_TMOUT : タイムアウトエラー LCD_E_PRM　: 引数異常 LCD_E_STS　: 状態異常
引数	1	型	LCD_CH_NO
		名称	ch_no
		コメント	チャネル番号
	2	型	LCD_CHARACTER
		名称	data
		コメント	書込み文字

```
LCDWrite( 0, 'S' );
```

図 8.2-21 LCD データ 1 文字書込み API 記述例

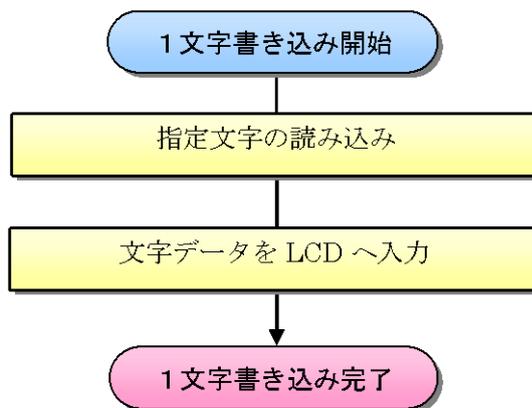



図 8.2-22 LCD データ 1 文字書込みのフローチャート

表 8.2-15 LCD データ文字列読み込み API　LCDReadLine

項目			内容
形式			LCD_RET LCDReadLine (LCD_CH_NO ch_no, LCD_CHARACTER *row, UINT8 line);
関数仕様			LCD のデータを 1 行読み込みます。 指定行の文字列を読み込み後、カーソルは元の位置に戻ります。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　　：正常終了 LCD_E_TMOUT：タイムアウトエラー LCD_E_PRM　：引数異常 LCD_E_STS　：状態異常
引数	1	型	LCD_CH_NO
		名称	ch_no
		コメント	チャネル番号
	2	型	LCD_CHARACTER *
		名称	row
		コメント	文字列格納アドレス(最低 16byte は領域を確保すること)
	3	型	UINT8
		名称	line
		コメント	行指定(1 or 2)

```
LCDReadLine( 0, str, 1 );
```

図 8.2-23 LCD データ文字列読み込み API 記述例

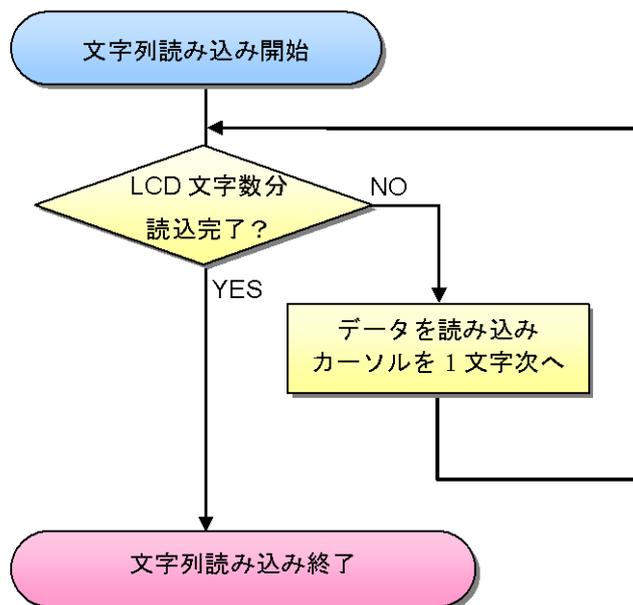



図 8.2-24 LCD データ文字列読み込みのフローチャート

表 8.2-16 LCD データ文字列書込み API　LCDWriteLine

項目			内容
形式			LCD_RET LCDWriteLine (LCD_CH_NO ch_no, LCD_CHARACTER *row, UINT8 line);
関数仕様			LCD へデータを 1 行書込みます。 指定行で文字列を書き込んだ後、カーソルは元の位置に戻ります。
戻り値	型		LCD_RET
	コメント		LCD_E_OK　　：正常終了 LCD_E_TMOUT：タイムアウトエラー LCD_E_PRM　：引数異常 LCD_E_STS　　：状態異常
引数	1	型	LCD_CH_NO
		名称	ch_no
		コメント	チャネル番号
	2	型	LCD_CHARACTER *
		名称	row
		コメント	書込み文字列の先頭アドレス
	3	型	UINT8
		名称	line
		コメント	行指定(1 or 2)

```
LCDWriteLine( 0, "Hello", 1 );
```

図 8.2-25 LCD データ文字列書込み API 記述例

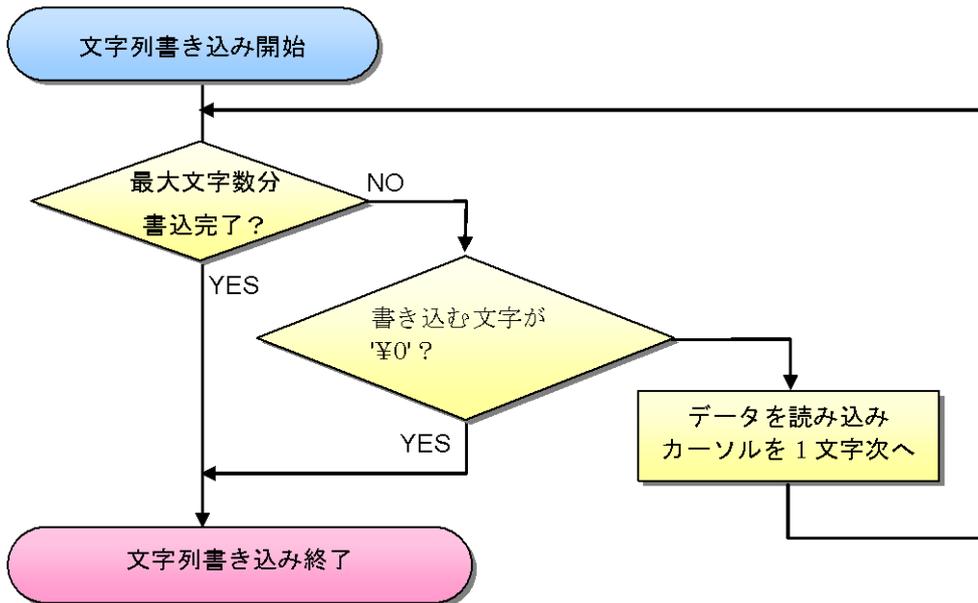



図 8.2-26 LCD データ文字列書込みのフローチャート

8.2.1.3 LED ドライバ API

LED ドライバ用の API を用いることで、LED 点灯/消灯の制御を行うことができます。**表 8.2-17～表 8.2-23** に LED ドライバ API の仕様を示します。また、**図 8.2-27～図 8.2-38**に API 呼出し記述例とフローチャートを示します。

使用する前に必ず初期化(LedInit)を一度呼出して下さい。

表 8.2-17LED ドライバ API 一覧

API 名	関数名	内容
LED 初期化 API	LedInit	LED を使用するための初期設定を行います。 LED を使用するためのポート出力設定を行います。
LED 終了 API	LedTerm	LED の使用を終了します。
LED 点灯 API	LedOn	指定の LED を点灯します。 引数 Led_No には LED2, LED3, LED4, LED5 のいずれかを指定します。複数指定も可能です。
LED 消灯 API	LedOff	指定の LED を消灯します。 引数 Led_No には LED2, LED3, LED4, LED5 のいずれかを指定します。複数指定も可能です。
LED 反転 API	LedRev	指定の LED の点灯/消灯を反転します。 引数 Led_No には LED2, LED3, LED4, LED5 のいずれかを指定します。複数指定も可能です。
LED 点灯／消灯状態参照 API	LedRef	LED2, LED3, LED4, LED5 の点灯状態を OR 演算したものを戻り値として返します。

表 8.2-18 LED 初期化 API　LedInit

項目			内容
形式			LED_RET LedInit (void);
関数仕様			LED を使用するための初期設定を行います。 LED を使用するためのポート出力設定を行います。
戻り値	型		LCD_RET
	コメント		LED_E_OK　：正常終了 LED_E_STS　：状態異常
引数	1	型	void
		名称	
		コメント	無し

```
LedInit();
```

図 8.2-27 LED 初期化 API 記述例

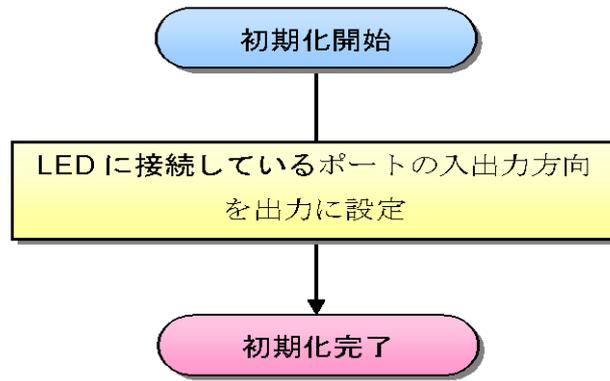



図 8.2-28 LED 初期化のフローチャート

表 8.2-19 LED 終了 API　LedTerm

項目			内容
形式			LED_RET LedTerm (void);
関数仕様			LED の使用を終了します。
戻り値	型		LCD_RET
	コメント		LED_E_OK　：正常終了 LED_E_STS　：状態異常
引数	1	型	void
		名称	
		コメント	無し

```
LedTerm();
```

図 8.2-29 LED 終了 API 記述例

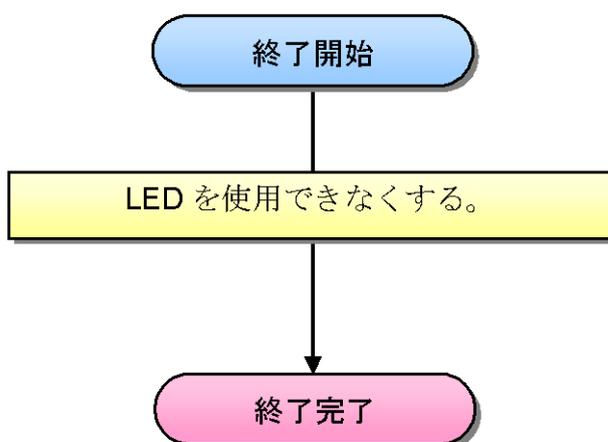



図 8.2-30 LED 終了のフローチャート

表 8.2-20 LED 点灯 API　LedOn

項目			内容
形式			LED_RET LedOn (UINT8 led_no);
関数仕様			指定の LED を点灯します。 引数 led_no には LED2, LED3, LED4, LED5 のいずれかを指定します。 複数指定も可能です。
戻り値	型		LCD_RET
	コメント		LED_E_OK　：正常終了 LED_E_STS　：状態異常 LED_E_PRM　：引数異常
引数	1	型	UINT8
		名称	led_no
		コメント	点灯する LED を指定します。 (LED2, LED3, LED4, LED5 のいずれか)

```
LedOn(LED2 | LED3);
```

図 8.2-31 LED 点灯 API 記述例

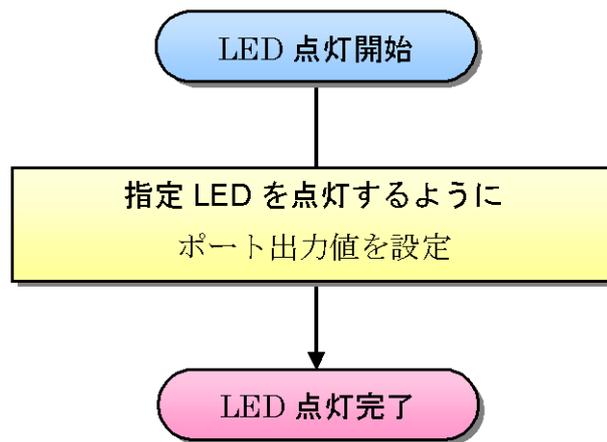



図 8.2-32 LED 点灯のフローチャート

表 8.2-21 LED 消灯 API　LedOff

項目			内容
形式			LED_RET LedOff (UINT8 led_no);
関数仕様			指定の LED を消灯します。 引数 led_no には LED2, LED3, LED4, LED5 のいずれかを指定します。 複数指定も可能です。
戻り値	型		LCD_RET
	コメント		LED_E_OK　：正常終了 LED_E_STS　：状態異常 LED_E_PRM　：引数異常
引数	1	型	UINT8
		名称	led_no
		コメント	消灯する LED を指定します。 (LED2, LED3, LED4, LED5 のいずれか)

```
LedOff(LED2 | LED3);
```

図 8.2-33 LED 消灯 API 記述例

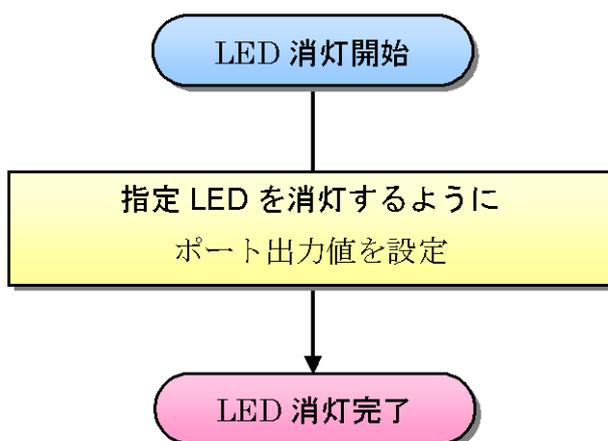



図 8.2-34 LED 消灯のフローチャート

表 8.2-22 LED 反転 API　LedRev

項目			内容
形式			LED_RET LedRev (UINT8 led_no);
関数仕様			指定の LED の点灯/消灯を反転します。 引数 led_no には LED2, LED3, LED4, LED5 のいずれかを指定します。 複数指定も可能です。
戻り値	型		LCD_RET
	コメント		LED_E_OK　：正常終了 LED_E_STS　：状態異常 LED_E_PRM　：引数異常
引数	1	型	UINT8
		名称	led_no
		コメント	反転する LED を指定します。 (LED2, LED3, LED4, LED5 のいずれか)

```
LedRev(LED2 | LED3);
```

図 8.2-35 LED 反転 API 記述例

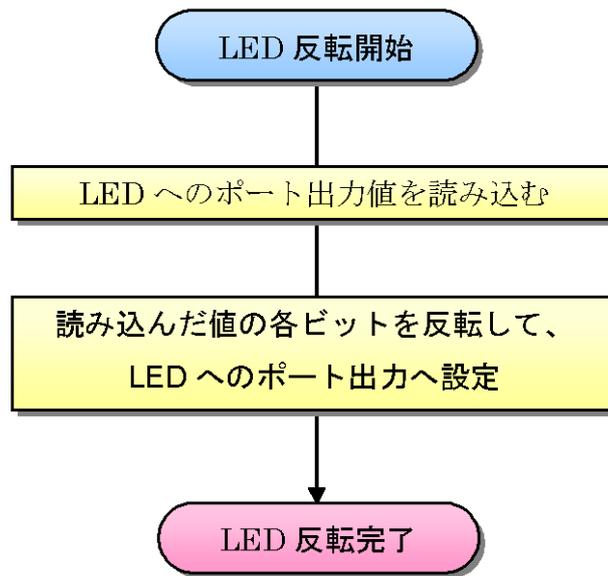



図 8.2-36 LED 反転のフローチャート

表 8.2-23 LED 点灯／消灯状態参照 API　LedRef

項目			内容
形式			UINT8 LedRef (void);
関数仕様			LED2, LED3, LED4, LED5 の点灯状態を OR 演算したものを 戻り値として返します。
戻り値	型		UINT8
	コメント		LED2, LED3, LED4, LED5 の点灯状態の OR 演算結果
引数	1	型	void
		名称	
		コメント	無し

```
UINT8 led_sts;
led_sts = LedRef();
```

図 8.2-37 LED 点灯／消灯状態参照 API 記述例

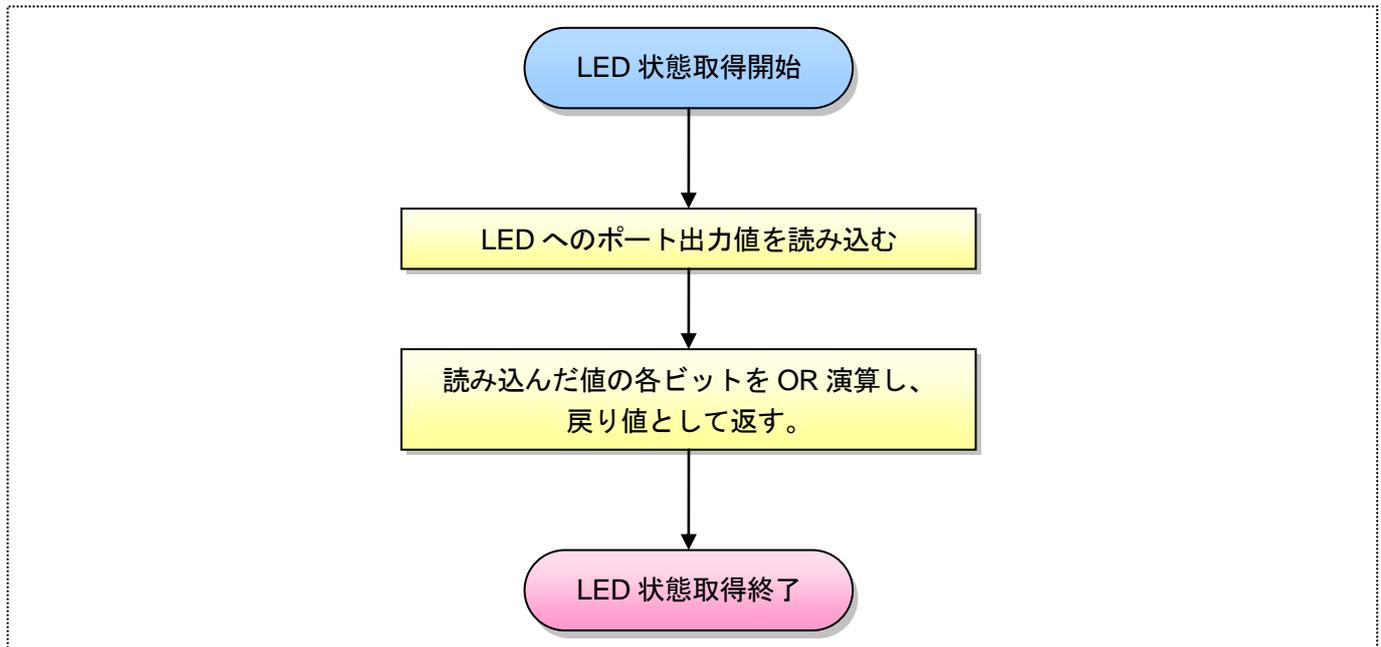



図 8.2-38 LED 点灯／消灯状態参照のフローチャート

8.2.1.4　　スイッチドライバ API

スイッチドライバ用の API を用いることで、スイッチ状態の読み込みを行うことができます。**表 8.2-24～表 8.2-30**に LED ドライバ API の仕様を示します。また、**図 8.2-39～図 8.2-44**に API 呼出し記述例とフローチャートを示します。

使用する前に必ず初期化(SwInit)を一度呼出して下さい。

表 8.2-24 スイッチドライバ API 一覧

API 名	関数名	内容
スイッチ初期化	SwInit	スイッチを使用するための初期設定を行います。
スイッチ終了	SwTerm	スイッチの使用を終了します。
SW3 ボタン状態取得	SwGetPush3	SW3 のボタン押下状態を取得します。
SW4 ボタン状態取得	SwGetPush4	SW4 のボタン押下状態を取得します。
SW5 ボタン状態取得	SwGetPush5	SW5 のボタン押下状態を取得します。
SW6 ボタン状態取得	SwGetPush6	SW6 のボタン押下状態を取得します。

表 8.2-25 スイッチ初期化 SwInit

項目			内容
形式			SW_RET SwInit(void);
関数仕様			スイッチを使用するための初期設定を行います。
戻り値	型		SW_RET
	コメント		SW_E_OK ： 正常終了 SW_E_STS ： 状態異常
引数	1	型	void
		名称	
		コメント	無し

```
SwInit();
```

図 8.2-39 スイッチ初期化 API 記述例

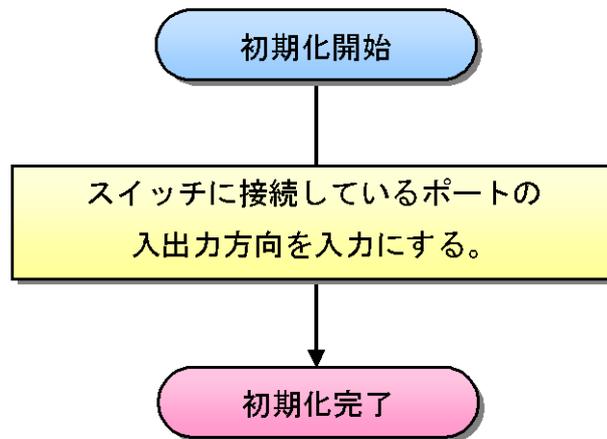



図 8.2-40 スイッチ初期化のフローチャート

表 8.2-26 スイッチ終了 SwTerm

項目			内容
形式			SW_RET SwTerm(void);
関数仕様			スイッチの使用を終了します。
戻り値	型		SW_RET
	コメント		SW_E_OK　：正常終了 SW_E_STS　：状態異常
引数	1	型	void
		名称	
		コメント	無し

```
SwTerm();
```

図 8.2-41 スイッチ終了 API 記述例

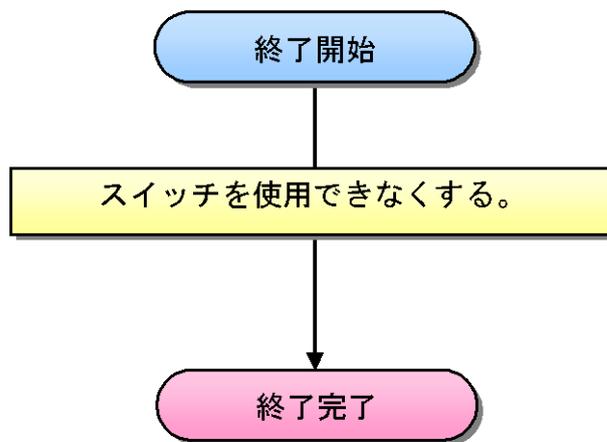



図 8.2-42 スイッチ終了のフローチャート

表 8.2-27 SW3 ボタン状態取得 SwGetPush3

項目			内容
形式			SW_RET SwGetPush3(UINT8 sw3_state);
関数仕様			SW3 のボタン押下状態を取得します。
戻り値	型		SW_ RET
	コメント		SW_E_OK　：正常終了 SW_E_STS　：状態異常
引数	1	型	UINT8 *
		名称	sw3_state
		コメント	SW3 のボタン押下状態の格納先を指定します。

表 8.2-28 SW4 ボタン状態取得 SwGetPush4

項目			内容
形式			SW_RET SwGetPush4(UINT8 *sw4_state);
関数仕様			SW4 のボタン押下状態を取得します。
戻り値	型		SW_RET
	コメント		SW_E_OK　：正常終了 SW_E_STS　：状態異常
引数	1	型	UINT8 *
		名称	sw4_state
		コメント	SW4 のボタン押下状態の格納先を指定します。

表 8.2-29 SW5 ボタン状態取得 SwGetPush5

項目			内容
形式			SW_RET SwGetPush5(UINT8 *sw5_state);
関数仕様			SW5 のボタン押下状態を取得します。
戻り値	型		SW_RET
	コメント		SW_E_OK　：正常終了 SW_E_STS　：状態異常
引数	1	型	UINT8 *
		名称	sw5_state
		コメント	SW5 のボタン押下状態を格納先を指定します。

表 8.2-30 SW6 ボタン状態取得 SwGetPush6

項目			内容
形式			SW_RET Sw_Get_Push6(UINT8 *sw6_state);
関数仕様			SW6 のボタン押下状態を取得します。
戻り値	型		SW_RET
	コメント		SW_E_OK　：正常終了 SW_E_STS　：状態異常
引数	1	型	UINT8 *
		名称	sw6_state
		コメント	SW6 のボタン押下状態の格納先を指定します。

```
SwGet_Push3( &sw3 );

SwGet_Push4( &sw4 );

SwGet_Push5( &sw5 );

SwGet_Push6( &sw6 );
```

図 8.2-43 スイッチ API 記述例

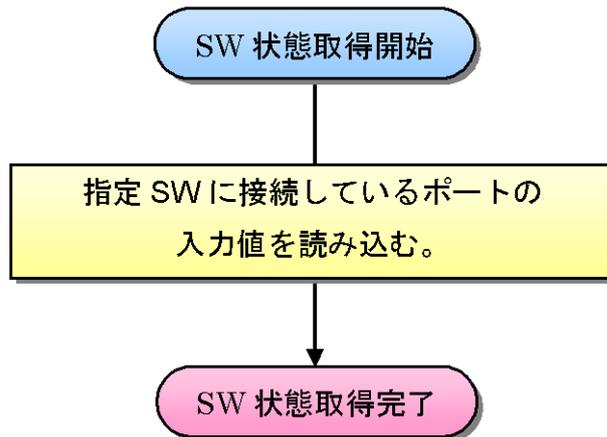



図 8.2-44 SW ボタン状態取得のフローチャート

※提供デバイスドライバの一つに A/D 変換デバイスドライバがありますが、これはリモコンキット用プログラムでのみ使用するデバイスドライバとなっています。

8.2.2 SFR

今回はあらかじめ作成されているデバイスドライバを提供しますが、実際にデバイスドライバを作成するには、周辺機器を制御するためのレジスタについて知っておく必要があります。そのレジスタは特殊機能レジスタ、略して SFR（Special Function Register）といい、デバイスドライバ内の処理では SFR を読み書きすることで周辺機器の操作をしています。

マイコンによって持っている機能や周辺機器が異なってきますので、SFR の内容も各マイコンで異なるものになります。デバイスドライバで抽象化されているのは、この SFR を読み書きする処理部分です。マイコンを変えた場合は SFR へのアクセス方法も合わせて変更しなければなりません。

例として、**表 8.2-3**で示したシリアル１文字入力 API（RecvPolSerialChar）では、シリアル用の SFR に書き込まれているデータを読み込む処理が行われています。シリアルの場合、一般的にはシリアルケーブルにデータが入力された時点で、入力された１バイトのデータがシリアル用 SFR にハードウェアが格納してくれますので、ソフトウェアとしてはそのデータを読み込むことになります。

ただし、マイコンによってデータが格納されているアドレス、データの持ち方、データが格納されるタイミング等が異なりますので、RecvPolSerialChar という API の中身もやはり各マイコン用に用意する必要があります。

このようにデバイスドライバ作成には、そのマイコンの SFR について知る必要がありますが、一般的に SFR の内容は各マイコンのハードウエアマニュアルに記載されています。デバイスドライバを作成する際には、実際に操作したい周辺機器用の SFR についてハードウエアマニュアルより調査するようにしましょう。

8.3 提供デバイスドライバを用いたアプリケーションの作成

8.2.1 提供デバイスドライバについてで学んだ LCD、LED、スイッチドライバを用いて、アプリケーションを作成してみましょう。

8.3.1 システム概要

本章で作成するデバイスドライバアプリケーションにおいて、スイッチ・LED・LCD は**図 8.3-1**のような関係になります。

スイッチを押している間は、対応した LED が点灯します。スイッチを離すとその LED は消灯します。

LCD は LED の点灯状態を見て、**図 8.3-1**で示すように出力します。また、LCD の初期表示状態は全て「OFF」を表示します。

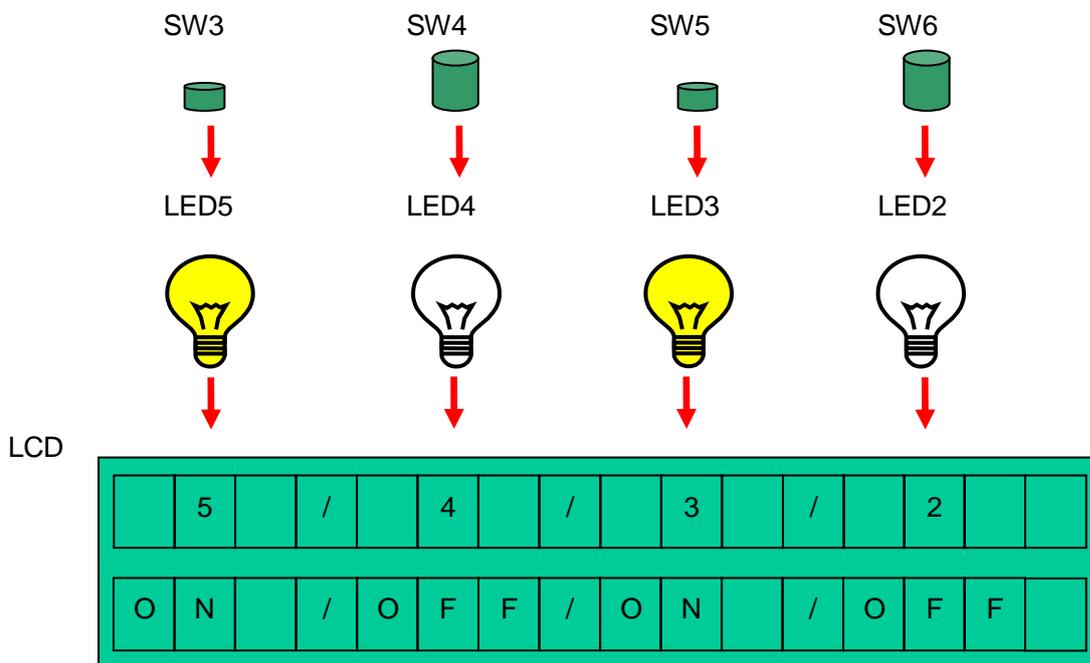



図 8.3-1 デバイスドライバアプリケーション システム概要

8.3.2 ソフトウェア仕様

8.3.2.1 タスク構成

本章で作成するデバイスドライバアプリケーションは、スイッチの押下状態を監視するスイッチ監視タスクと、LED・LCD の表示を行う表示タスクの 2 種類のタスクから構成されます。

また、初期化処理に関しては StartupHook 内で行われます。

図 8.3-2はアプリケーションの全体処理フローです。

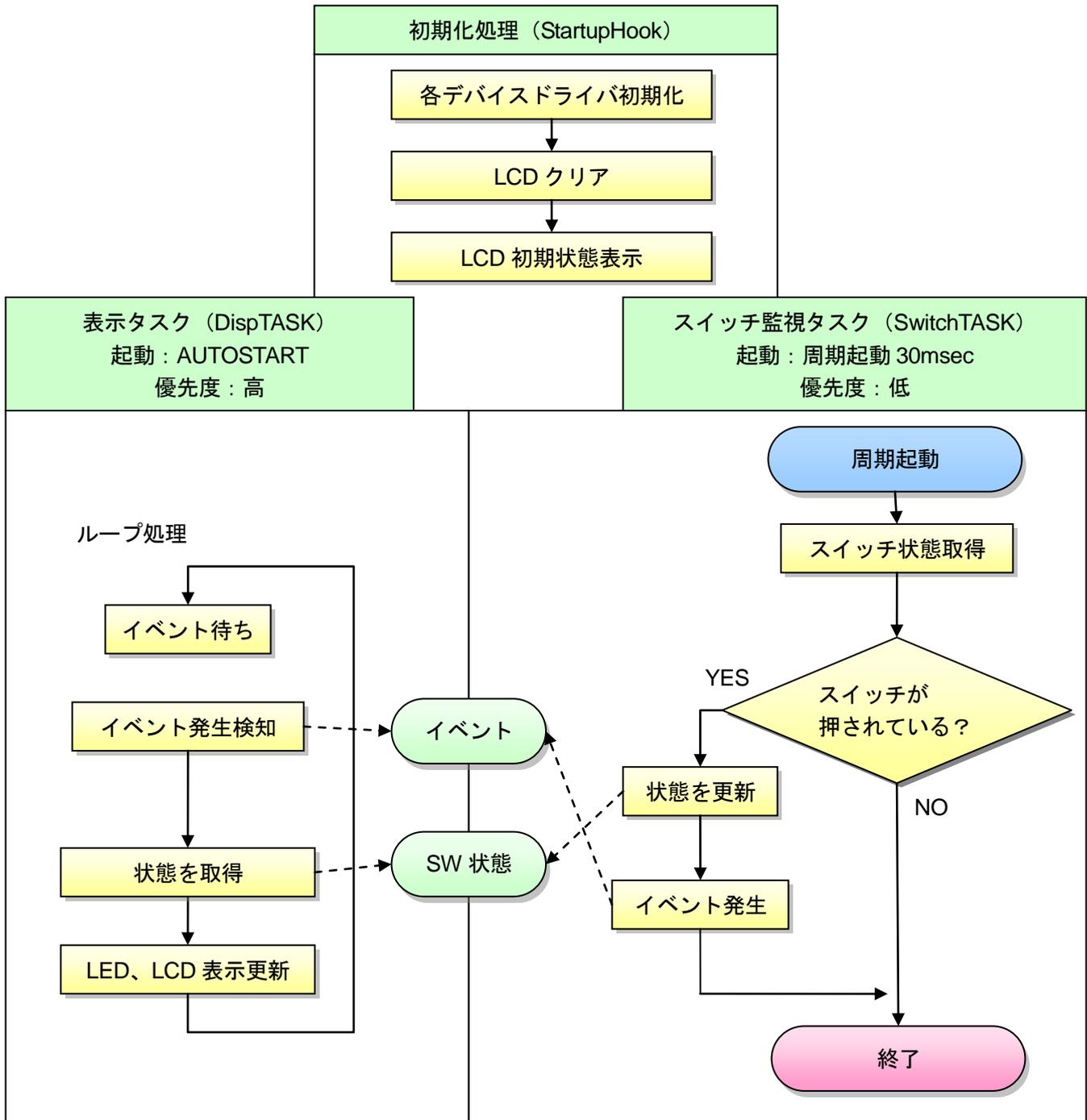



図 8.3-2 デバイスドライバアプリケーション 全体処理フロー

8.3.2.2 OIL オブジェクト情報

表 8.3-1はスイッチ監視タスクのオブジェクト設定情報、**表 8.3-2**は表示タスクのオブジェクト設定情報となります。

表 8.3-1 SwitchTASK の設定情報

タスク名		SwitchTASK
タスク属性値	優先度(PRIORITY)	6
	スケジューリング(SCHEDULE)	FULL
	起動可能数(ACTIVATION)	1
	自動実行(AUTOSTART)	FALSE

表 8.3-2 DispTASK の設定情報

タスク名		DispTASK
タスク属性値	優先度(PRIORITY)	7
	スケジューリング(SCHEDULE)	FULL
	起動可能数(ACTIVATION)	1
	自動実行(AUTOSTART)	TRUE

また、**表 8.3-3**はスイッチ監視タスクを起動する周期アラームのオブジェクト設定情報になります。

表 8.3-3 ARARM オブジェクトの設定情報

属性	設定値
COUNTER	SysTimerCnt

ACTION	ACTIVATETASK {
	TASK = SwitchTASK
	}

AUTOSTART	TRUE {
	APPMODE = AppMode1;
	ALARMTIME = 999;
	CYCLETIME = 30;
	}

演習問題

システム概要、ソフトウェア仕様を参照して、デバイスドライバアプリケーションを作成して下さい。
また、必要なオブジェクトを OIL ファイルに追記して下さい。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex8_led_lcd」フォルダにセットアップ済みです。

①「H25_toppers_ ex8_led_lcd」フォルダをコピー
②「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

CAN 通信

9.1 CAN 通信プロトコル

本章では､操作に対する多様性や即応性が求められるエンジン制御やハンドル操作といったエンジン･パワートレイン系の制御に用いられる CAN について説明します。

9.1.1 CAN コントローラ

CAN のコントローラは送受信用のバッファを複数個持っています。このバッファの数が少ないタイプのコントローラを BasicCAN、多いタイプを FullCAN といいます。FullCAN は BasicCAN に比べ処理が早く、高速通信で使用されます。BasicCAN はバッファが少ない分コストが削減できます。

9.1.2 コントローラフォーマット

CAN のコントローラフォーマットは標準フォーマット 2.0A､拡張フォーマット 2.0B パッシブ､拡張フォーマット 2.0B アクティブの 3 種類があります。

CAN には各ノードの識別やメッセージの優先順位の判定を行うための ID（詳細後述）が存在し、標準フォーマット（11 ビット）と拡張フォーマット（29 ビット）の 2 種類のフォーマットがあります。

2.0A では標準フォーマットのみの送受信が可能です。もし拡張フォーマットを受信してしまった場合、エラーが発生してしまいます。

2.0B パッシブは、送受信が可能なのは 2.0A と同じ標準フォーマットのみですが、拡張フォーマットを受信した場合、それを無視することでエラーを回避することができます。

2.0B アクティブは標準フォーマットも拡張フォーマットも送受信が可能です。

現在はより多くの ID を使用できる 2.0B アクティブが主流となっています。

ID / プロトコル	標準フォーマット（11 ビット）		拡張フォーマット（29 ビット）	
	送信	受信	送信	受信
2.0A	○	○	×（送信不可）	×（エラー発生）
2.0B パッシブ	○	○	×（送信不可）	△（無視）
2.0B アクティブ	○	○	○	○

図 9.1-1 CAN コントローラフォーマット一覧

9.1.3 通信バス規格

CAN の通信バスの規格には最大 1Mbps の High Speed CAN と、最大 125kbps の Low Speed CAN という 2 種類があります。

両方の規格とも、ノイズの影響を受けにくくするため 2 本の通信線を使用しています。しかし、Low Speed CAN は High Speed CAN に比べて通信速度が遅い代わりに、通信線に問題が発生した場合、1 本の線のみで通信を行うことができます。

2 種類の規格の差異は CAN コントローラが吸収してくれるため、混在して使用しても問題ありません。

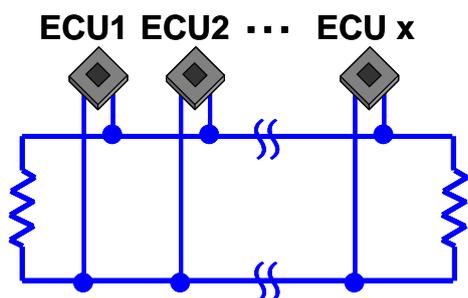
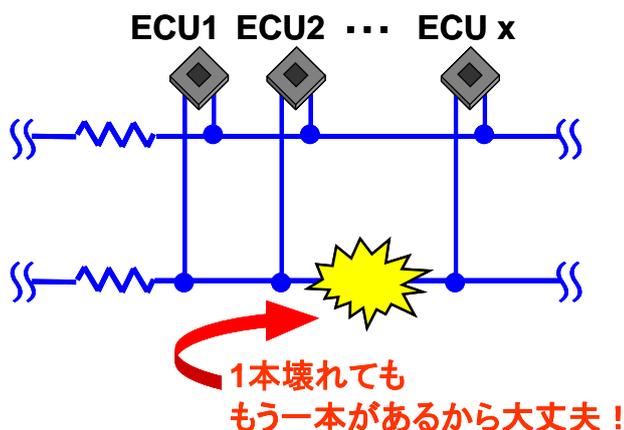



図 9.1-2 High Speed CAN と Low Speed CAN

9.1.4 ドミナントとリセッシブ

デジタル通信においてすべてのデータは 0 と 1 で表される 2 進数に変換されて通信されます。

CAN では 0 をドミナント（Dominant：優性）、1 をリセッシブ（Resessive：劣性）といいます。

通信において、複数のノードからドミナントとリセッシブが同時に送信された場合、言葉の意味通りドミナントが優先されることとなります。

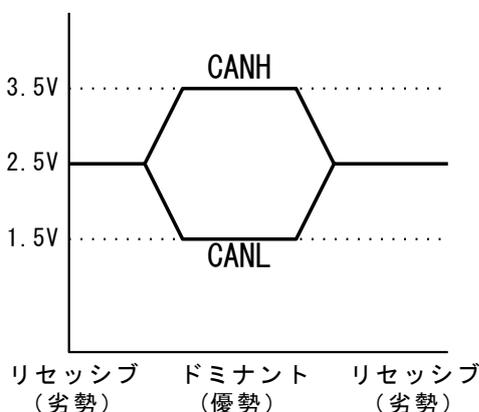



図 9.1-3 High Speed CAN と Low Speed CAN

9.1.5 同期とスタッフィングルール

CAN は通信データがリセッシブ（1）→ドミナント（0）に変化するときに同期をとっています。

しかし、通信データが長い間同じ状態になる可能性も当然あります。そうすると、その間は同期がとれなくなってしまい、通信のタイミングが徐々にずれていってしまいます。

このような問題を回避するために、CAN ではスタッフィングルールという規則を設けています。

これは、同じ通信データが 5 ビット続いたとき、その次のデータにスタッフビットという反転ビットを入れるというルールです。こうすることにより、6 ビット以上同じ通信データが続かないようにしています。

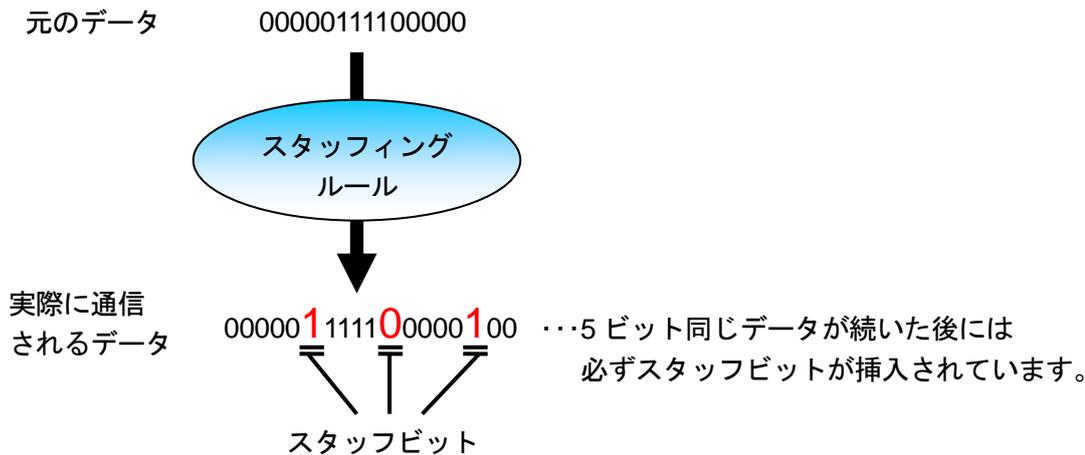



図 9.1-4 スタッフィングルール

9.1.6 マルチマスタ方式 ／ イベントドリブン方式

マルチマスタ方式とは､バスが空いていればどのノードでもメッセージを送信することができる通信方式です。また、各 ECU でネットワーク管理を行うので、ノードの自由な着脱が可能となっています。

イベントドリブン方式とは、指定されたイベント（スイッチが押された、特定のデータを受信した等）に対応して処理を行うという実行方式のことです。

つまり､CAN は指定したイベントに対応してデータの送信処理を行う､自由度の高い通信方式となっています。

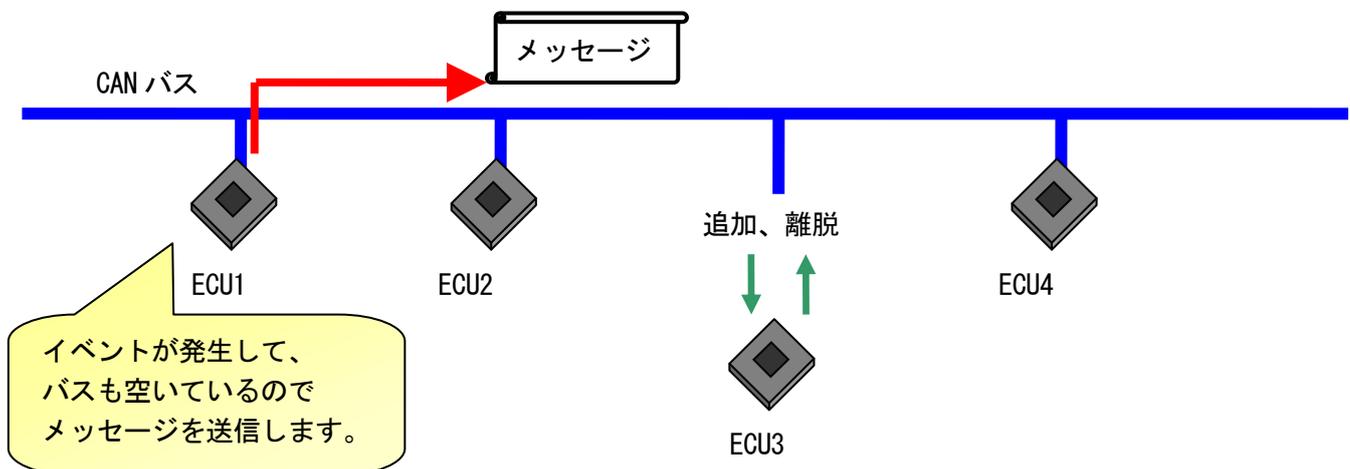



図 9.1-5 マルチマスタ方式 ／ イベントドリブン方式

9.1.7 アービトレーション（調停）

マルチマスタ方式とイベントドリブン方式を組み合わせた通信方式は自由度が高い一方で、複数の ECU が同時に送信処理を行ってしまい、メッセージの衝突が発生する危険性も含んでいます。一般的なネットワークでは、衝突が起きた場合、通信をストップし、ある程度の時間を置いた後で通信を再開するという方式をとっていますが、その後繰返し衝突が発生し、長い間送信が出来ない状態が続く可能性があります。通信ができない可能性があるということは、すなわち通信の信頼性の低下を意味します。

CAN プロトコルでは信頼性の向上のため、衝突が発生した場合、アービトレーション（調停）という処理を行います。メッセージに含まれる識別子（Identifier : ID）を使用して、衝突したノードのいずれかに送信優先権を与えることで、通信をストップさせることがないようにしています（**図 9.1-6**）。

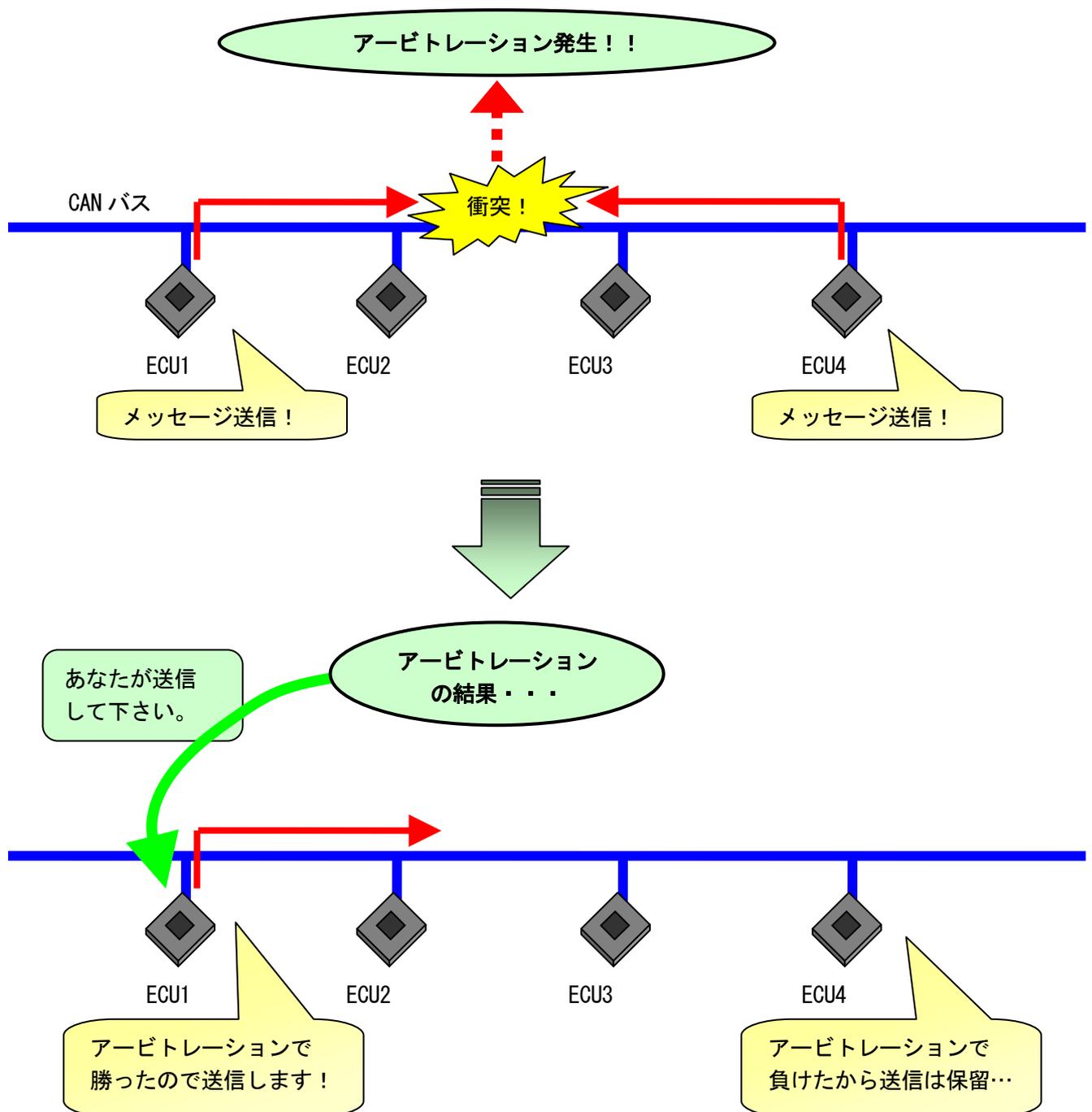



図 9.1-6 アービトレーション

9.1.8 フレーム

CAN は「データフレーム」「リモートフレーム」「エラーフレーム」「オーバーロードフレーム」と呼ばれる 4 種類のフレームを用いて通信を行います。

9.1.8.1　データフレーム

データフレームはデータを送信する転送フォーマットです。データフレームのフォーマットは標準フォーマットと拡張フォーマットの 2 種類があります。

以下は標準フォーマットのデータフレーム構造です。

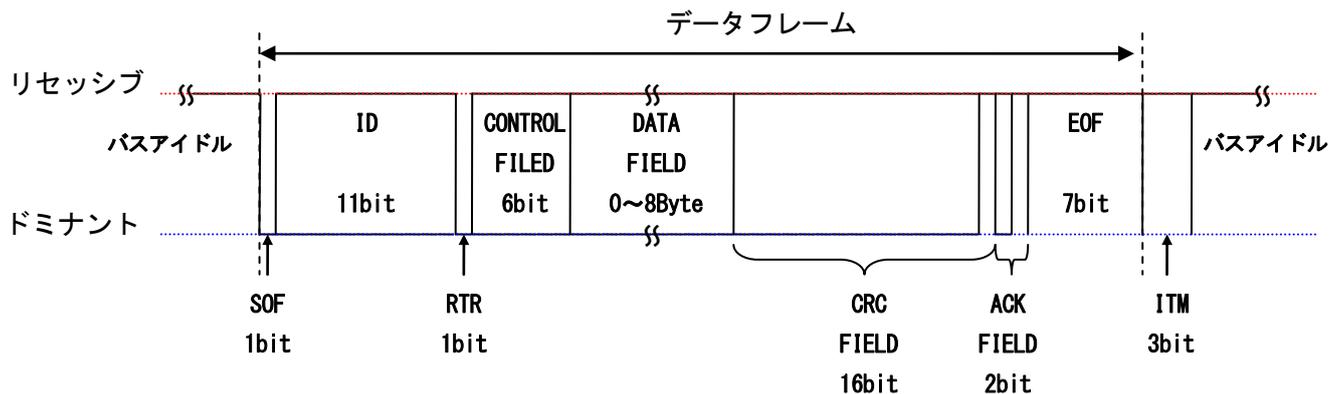



図 9.1-7 データフレーム（標準フォーマット）

- SOF（Start Of Frame）：バスアイドル状態であるリセッシブからドミナントに変化することで、データフレームの送信を表し、受信側と同期をとります。

- ID（Identifier）：データ内容や送信ノードの識別や、先述したアービトレーションで優先順位を決定するのに用いられます。11 ビット長で構成されます。

- RTR（Remote Transmission Request）：データフレームとリモートフレームの間に入り、両者の区別をします。RTR は必ずドミナントとなります。

- コントロールフィールド：予約ビット r0、r1 と 4 ビットの DLC（Data Length Code）から構成されます。DLC は後述のデータフィールドが何バイトになるかを示します。

- データフィールド：実際の送信データです。データ長は前述のコントロールフィールドで決定されます。

- CRC（Cyaclic Redundancy Check）フィールド：CRC シーケンスと CRC デリミタから構成されます。CRC シーケンスでは、SOF、ID、コントロールフィールド、データフィールドから演算した値が送信されます。受信側は受信した SOF、ID、コントロールフィールド、データフィールドを同じように演算し、CRC シーケンスの値と比較することで、正常にデータが受信できたかを判別します。
CRC デリミタは CRC シーケンスの終了を表し、必ずリセッシブとなります。

- ACK（Acknowledgement）フィールド：ACK スロットと ACK デリミタから構成されます。
ACK スロットでは送信ノードは必ずリセッシブの送信を行います。対して、受信ノードは CRC フィールドまで正常に受信できていた場合、その確認応答として ACK スロットのタイミングでドミナントを送信します。CAN では、同じタイミングでリセッシブとドミナントが送信された場合、ドミナントが優先されます。つまり、ACK スロットのタイミングでドミナントとなっているということは、正常にデータが受信できた受信ノードが存在することを意味します。

ACK デリミタは ACK スロットの終了を表し、必ずリセッシブとなります。

・ EOF（End of Frame）: データフレームの終了を表し、必ずリセッシブの 7 ビット長となります。

最後の ITM（intermission）はフレームではありません。リセッシブの 3 ビットを送信し、その後バスアイドル状態となることを示します。

拡張フォーマットでは、標準フォーマットの ID11 ビットに拡張 ID18 ビットが追加され、合計 29 ビット長の ID を表すことができます。

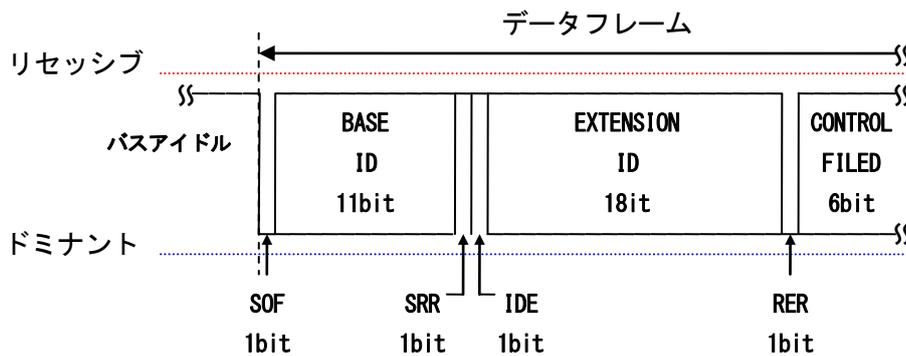
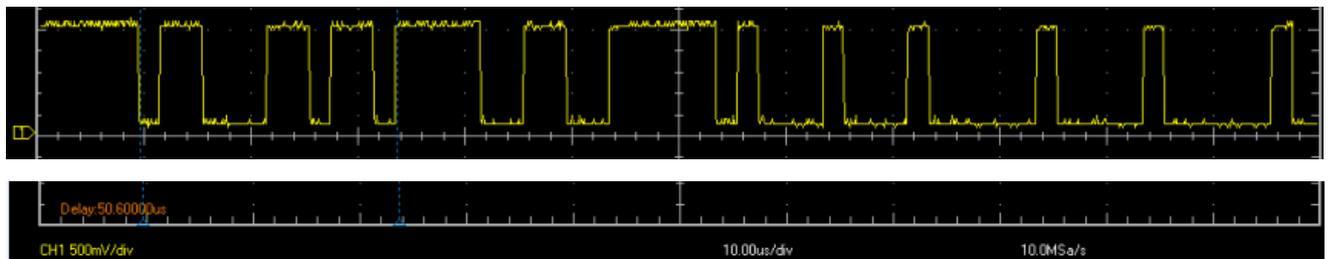



図 9.1-8 データフレーム（拡張フォーマット）

標準フォーマットで ID だったフィールドは、拡張フォーマットでは BASE ID といいます。
BASE ID の後、リセッシブ 1 ビット長の SRR（Substitute Remote Request Bit）と IDE（Identifier Extention Bit）が続きます。
その後に拡張 ID である EXTENSION ID が 18 ビット長送信されます。
CONTROL FIELD 以降は標準フォーマットと変わりません。

演習問題

下記波形は、転送速度が 500Kbps／拡張ＩＤ（29 ビット）、データフレームの波形です。
さて、ＩＤ値はいくつでしょうか？

9.1.8.2　　リモートフレーム

リモートフレームは特定のノードにデータフレームの要求をする際に用います。

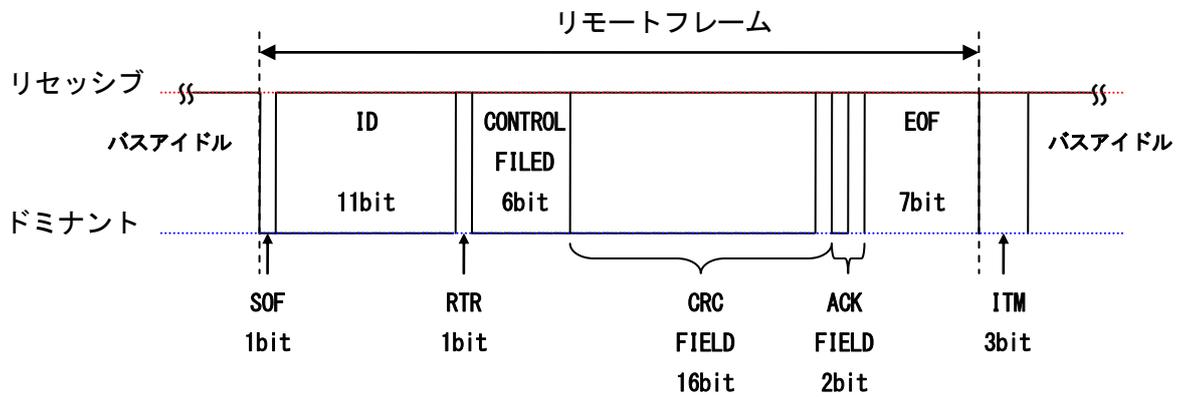



図 9.1-9 リモートフレーム

基本的なフレーム構造は、データフレームからデータフィールドを除いたものになります。

データフレームとの区別は RTR で行います。データフレームではドミナントとなる RTR ですが、リモートフレームではリセッシブになります。

ID に要求するデータフレームの ID、コントロールフィールドの DLC に要求するデータフレームのデータ長を設定して使用します。

9.1.8.3　オーバーロードフレーム

オーバーロードフレームは CAN コントローラが前回のフレームの処理を終了させるための待ち時間を作るために用いられます（昔のマイコンは処理能力が低く、前回のフレームの処理が間に合わない場合があったため発案されたフレームです。現在のマイコンは昔に比べ処理能力が上がっているため、処理が間に合わないということはほぼありませんが、CAN の規格上オーバーロードフレームの機能は持っています）。

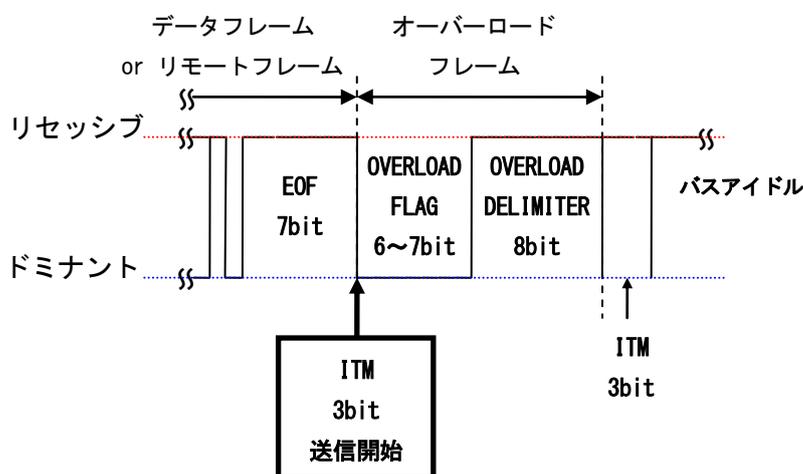



図 9.1-10 オーバーロードフレーム

- オーバーロードフラグ：ドミナントの 6 ビット長でオーバーロードフレームが送信されていることを示します。ノードが送信するビット長は 6 ビットですが、送信動作により CAN バスには 7 ビット現れることもあります。
- オーバーロードデリミタ：リセッシブの 8 ビット長でオーバーロードフレームの終了を示します。

データフレーム、もしくはリモートフレームが送信終了（EOF 送信終了）し、ITM を送信開始するタイミングで、処理待ちが必要なノードがオーバーロードフラグを 2 ビット以内に送信します（3 ビット＝ITM がすべて送信されるとバスアイドルになってしまうため 2 ビット以内に送信します）。

ITM のバスレベルがリセッシブに対して、オーバーロードフラグはドミナントなので、オーバーロードフラグが優先されます。

オーバーロードフラグを受け取った他のノードはすぐにオーバーロードフラグを送り返します。つまり、最初のオーバーロードフラグから 1 ビット遅れで送信するため､CAN バスには計 7 ビット長のオーバーロードフラグが現れます。もし、すべてのノードが ITM 送信のタイミングでオーバーロードフラグを送信した場合、送り返しがないため、6 ビット長になります。

オーバーロードフラグ送信後、オーバーロードデリミタと ITM を送信しバスアイドルになります。

このように、ITM にオーバーロードフラグをかぶせて送信することで、バスアイドル状態に遷移する時間を遅くし、処理時間を稼ぐことができます。

9.1.8.4　エラーフレーム

エラーフレームは通信エラーが発生した際に送信されるフレームです。

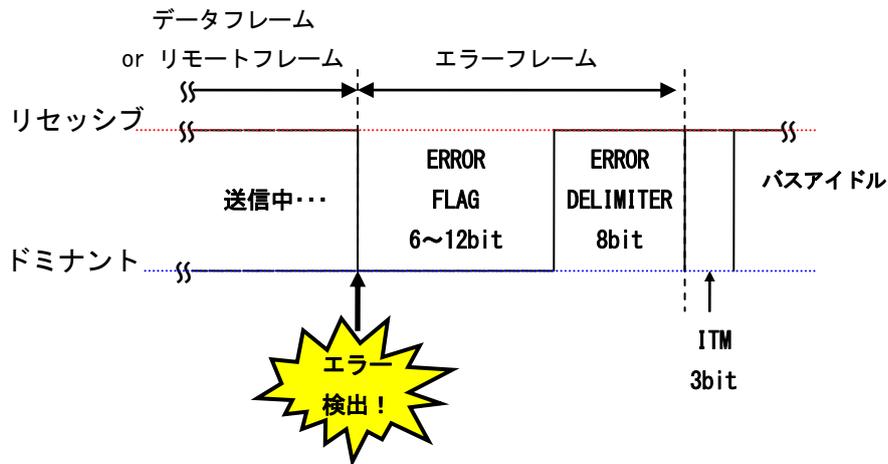



図 9.1-3 エラーフレーム

- エラーフラグ：ドミナントの 6 ビット長で、エラーが発生したことを示します。
- エラーデリミタ：リセッシブの 8 ビット長でエラーフレームの終了を示します。

データフレームもしくはリモートフレームの送受信にエラーが検出された場合、そのノードはエラーフラグを送信します。そのとき、わざとスタッフィングルールを守らずに（6 ビット同じデータを送り）、スタッフィングエラーを発生させます。

他のノードはこのスタッフィングエラーによって、エラーが発生したことを判断し、エラーフラグを送り返します。

エラーを検出したノードからのエラーフラグと、スタッフィングエラーを受けて返されたエラーフラグは CAN バス上で重複する場合があるため、CAN バスには 6～12 ビットのエラーフラグが現れます。

エラーの内容はエラーフレームが送られたタイミングで判別され、以下の 5 種類が存在します。

- ビットエラー：送信ノードが送信した内容と実際にバスに流れている内容が異なった場合に検出するエラーです。送信ノードで検出されます。
- ACK エラー：ACK スロットで、どの受信ノードもドミナントを返さなかった場合に検出するエラーです。すべての受信ノードが誤ったメッセージを受信したか、他のノードが一つも接続されていない場合に検出されます。送信ノードで検出されます。
- CRC エラー：受信したメッセージと CRC シーケンスの値が異なる場合に検出されるエラーです。受信ノードで検出されます。
- スタッフエラー：ビットスタッフィングルールが守られなかった場合に検出されるエラーです。受信ノードで検出されます。
- フォーマットエラー：本来リセッシブであるはずの CRC デリミタ、ACK デリミタ、EOF がドミナントになっている場合に検出されるエラーです。受信ノードで検出されます。

9.1.9 エラー状態

CAN には 3 つのエラー状態があります。

・ エラーアクティブ状態：通信に正常に参加できる状態です。

・ エラーパッシブ状態：通信には参加できますが、エラーを起こしやすいと判断された状態です。エラーを起こしやすい状態なので、もしエラーパッシブ状態のノードが繰返しエラーを出し続けてしまったら、他の正常なノードの通信の妨げになってしまいますので、エラーパッシブ状態とエラーアクティブ状態では送受信の処理が異なります。
  送信時には優先的にエラーアクティブ状態のノードから送信させるために送信待機を行います。ITM を受信した後、さらに 8 ビットのリセッシブ（バスアイドルを示すリセッシブ）を受信してからでないと送信処理を行うことができません。
  また、エラーが発生したときに送信するエラーフラグは通常ドミナントですが、エラーパッシブ状態のノードが送信するエラーフラグはリセッシブになります。受信時には送信ノードを含む他のノードが通信を行うため、リセッシブであるエラーフラグは無視されることとなり、結果エラーパッシブ状態の受信時エラーは他の通信を妨げません。
  しかし、送信をするとき、通信するのは一つのノードだけなので、リセッシブでも 6 ビット続いた時点でスタッフィングエラーを検出します。このことより、エラーパッシブ状態のノードが発生させるエラーは送信時のエラーだけということになります。

・ バスオフ状態：通信に参加できない状態です。

これら 3 つの状態は、送信エラーカウンタと受信エラーカウンタによって管理されています。各エラーカウンタはエラー時に増加し、正常通信時に減少します。図 9.1-4はエラー状態の遷移図です。

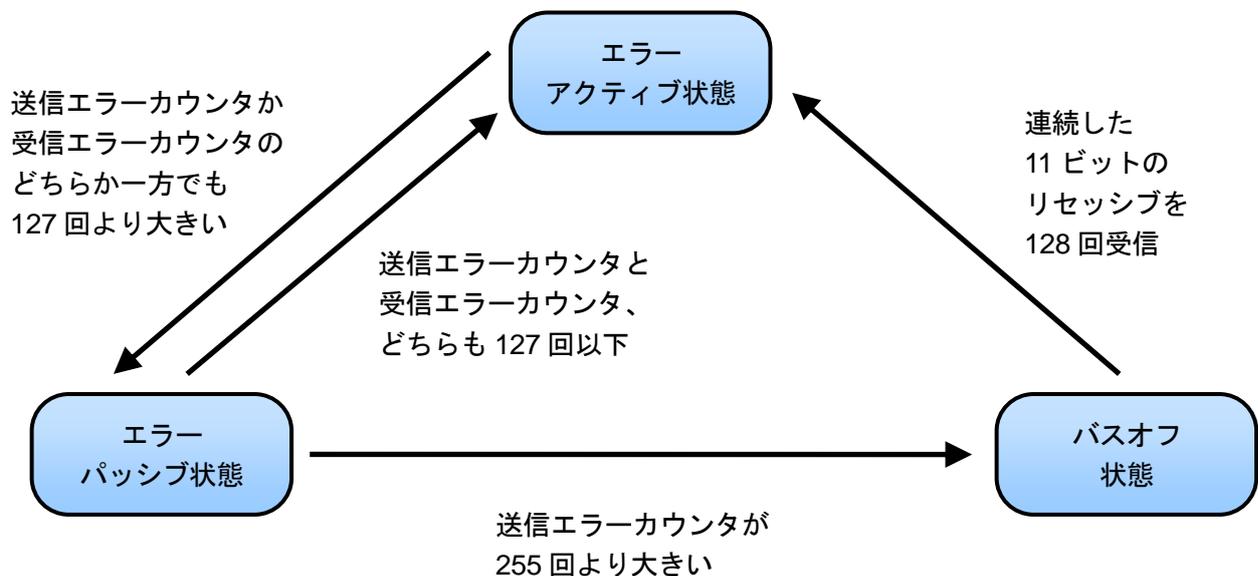



図 9.1-4 エラー状態遷移図

9.1.10　ビットタイミング

データの最小単位は 1 ビットですが、CAN はこの 1 ビットをさらに 4 つのセグメント（区分）に分けています。このセグメントは TimeQuantum（以下 Tq）という単位で表されます。ビットをセグメントと Tq で分割・構成することをビットタイミングといいます。

9.1.5 同期とスタッフィングルールで、CAN はリセッシブ（1）→ドミナント（0）のビットデータの切り替えで同期をとっているとしましたが、色々な遅延や誤差により、送信ノードが送信したタイミングと受信ノードが受信したタイミングがずれることがあります。このずれを修正し再同期させるために、このビットタイミングを用います。

表 9.1-1は各セグメントの役割と Tq 数を表します。

表 9.1-1 セグメントの役割と Tq 数

セグメント名	役割	Tq 数
Synchronization Segment（SS）	各ノードが通信タイミングを合わせるためのセグメントです。 このセグメントでリセッシブ⇔ドミナントのエッジ切り替えが行われるようにすることで、同期を行います。	1Tq
Propagation Time Segment（PTS）	ノードにおける送受信の入出力遅延や、CAN バスにおける転送遅延など、通信における物理的な遅延を修正するためのセグメントです。	1～8Tq
Phase Buffer Segment1（PBS1）	各ノードは各々が持つ周波数を元に動作していますが、その周波数に誤差が発生した場合、それを修正するためのセグメントです。 エッジ切り替えに対して SS が進んだ場合は PBS1 を増加、SS が遅れた場合は PBS2 を減少させて、SS の調整をします。	1～8Tq
Phase Buffer Segment2（PBS2）		2～8Tq

他に、reSynchronization Jump Width（SJW）という、PBS1,2 を増減・減少させる値があり、1～4Tq の範囲で設定できます。

ビットタイミングによってノード間の確実な同期を行うため、ビットタイミングの設定はすべてのノードで統一させる必要があります。

各セグメントの Tq を長くすると、遅延や誤差に対する修正の幅が広がるため、同期ミスの確率が減少します。しかし一方で、1 ビットの長さが長くなってしまうため、転送速度が遅くなってしまいます。

転送速度を取るのか、同期ミスの確率を減らすのかはどのような CAN ネットワークを作るかによって変わります。

9.1.11　アクセプタンスフィルタ

CAN の送信メッセージはすべての受信ノードに送信されます。しかし、すべての受信ノードがそのメッセージを必要としているとは限りません。受信ノードが必要なデータのみを受信するための機能がアクセプタンスフィルタです。

必要なデータであるかを判断するにはデータフレームの ID を使用します。受信ノードは自分が受信したいメッセージの ID を設定しておき、送信メッセージの ID がそれと一致した場合のみメッセージを受信します。

逆を言えば、アクセプタンスフィルタを使用しない（アクセプタンスマスク）ようにすると、複数のメッセージが受信できるようにすることも可能です。

9.2 CAN 通信ドライバの開発

本章では、**9.1 CAN 通信プロトコル**で学んだ CAN ドライバを実際に作成します。

9.2.1 CAN デバイスドライバ構成

本章では CAN デバイスドライバを**図 9.2-1**の構成で作成します。

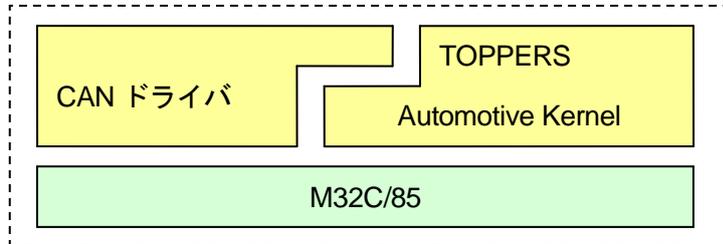



図 9.2-1 CAN デバイスドライバ構成

また、本章で作成する CAN ドライバは**5 TOPPERS Automotive Kernel の使用方法**で説明した**表 9.2-1**の環境で作成をします。

表 9.2-1 TOPPERS Automotive Kernel 実装環境

種類	製品名	メーカ名
統合開発環境	High-performance Embedded Workshop	株式会社ルネサステクノロジ
開発環境(コンパイラ等)	M3T-NC308WA	株式会社ルネサステクノロジ
実装対象ハードウェア	TOPPERS Platform ボード	株式会社サニー技研
リアルタイム OS	TOPPERS Automotive Kernel	TOPPERS プロジェクト
書込みソフトウェア	E8a エミュレータ	株式会社ルネサステクノロジ
シミュレータ	M32C シミュレータデバッガ	株式会社ルネサステクノロジ

表 9.2-2は、ターゲットとなるマイコン M32C/85 が持つ CAN コントローラの機能の仕様と、今回作成するデバイスドライバで使用する機能及び設定の一覧です。

表 9.2-2 CAN コントローラ機能及びデバイスドライバ実装機能・設定一覧

CAN コントローラ搭載機能	デバイスドライバ実装機能及び設定
CAN チャネルを 2 つ搭載	CAN0 チャネルを使用します。 （本ボードは CAN0 チャネルのみ端子が搭載されています。） CAN0 出力ポートは P7_6、CAN0 入力ポートは P7_7 を使用します。
メッセージスロットを 16 本（0〜15）搭載	メッセージスロット 0 を送信用、 メッセージスロット1〜15 を受信用として使用します。
メッセージスロットバッファを 2 つ搭載 （メッセージスロットにアクセスするときは、 このバッファを通してアクセスします。）	バッファ 0 にはメッセージスロット 0 を、 バッファ 1 にはメッセージスロット 1〜15 を割り当てます。
標準 ID か拡張 ID、どちらかを選択できます。	今回の演習ではたくさんのノードを用いないため、 標準 ID を使用します。
アクセプタンスフィルタ マスクレジスタでフィルタの設定ができます。 ・ グローバルマスクレジスタ ・・・メッセージスロット 0〜13 用 ・ ローカルマスク A ・・・メッセージスロット 14 用 ・ ローカルマスク B ・・・メッセージスロット 15 用	アクセプタンスフィルタを使用します。
転送速度 最大転送速度 1Mbps	転送速度を 500Kbps に設定します。 ビットタイミングは、 PTS ：5Tq PBS1：6Tq PBS2：4Tq SJW ：3Tq サンプリング回数：3 回 と設定します。
エラー検出 ビットエラー、ACK エラー、CRC エラー、 スタッフエラー、フォーマットエラーの検出が 可能です	エラーが発生しても、その通知を行うことはしません。
割込み 送信終了時、受信終了時、エラー発生時に 割込みを発生させることが出来ます。	受信終了の割込みを使用します。
タイムスタンプ機能	使用しませんが、タイムスタンププリスケーラは念のためCANバスビットクロックで初期化設定しておきます。
リモートフレーム自動応答機能	使用しません。
送信アボート機能	使用しません。
ループバック機能	使用しません。
エラーアクティブ強制復帰機能	使用しません。
シングルショット送信機能	使用しません。
自己診断機能	使用しません。

9.2.2 製作の環境準備

プロジェクトの driver フォルダに**図 9.2-2**のようにフォルダ及びファイルを作成して下さい。

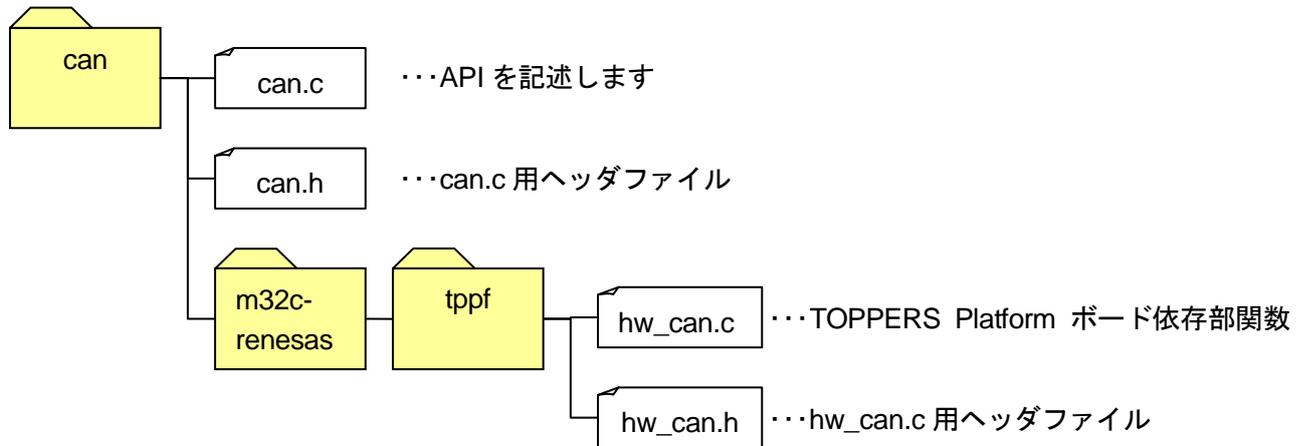



図 9.2-2 ディレクトリ・ファイル構成

5.2 プログラム作成手順を参考に上記のフォルダ・ファイルをプロジェクトに登録して下さい。

9.2.3 コールバック

たとえば何かのデータの受信処理をする場合、受信している間ずっと待っていると、上位層はその間処理が全くできなくなり、非常に無駄な時間が生じてしまいます。この無駄な待ち時間を防ぐために、受信処理は、受信の要求処理だけで、受信データの完了自体は割込みを使って上位層に伝える手法があります。これをコールバックといいます。

また、コールバックをするために使用する関数をコールバック関数といいます。これはデバイスドライバを使用する際に、上位層に必ず作ってもらう関数で、これを呼び出すことで完了通知とします。また、その関数に引数を用意すれば、完了通知と一緒にデータの引渡しも可能になります。

受信して
下さい！
受信完了まで
他の処理を実行
では、受信後の
処理します
上位層
コールバック
ルーチン
受信要求
あとで
連絡するよ
受信完了
通知
受信
終わったよ
デバドラ
デバドラ
API
受信完了
関数
受信要求
受信完了割込み
デバイス

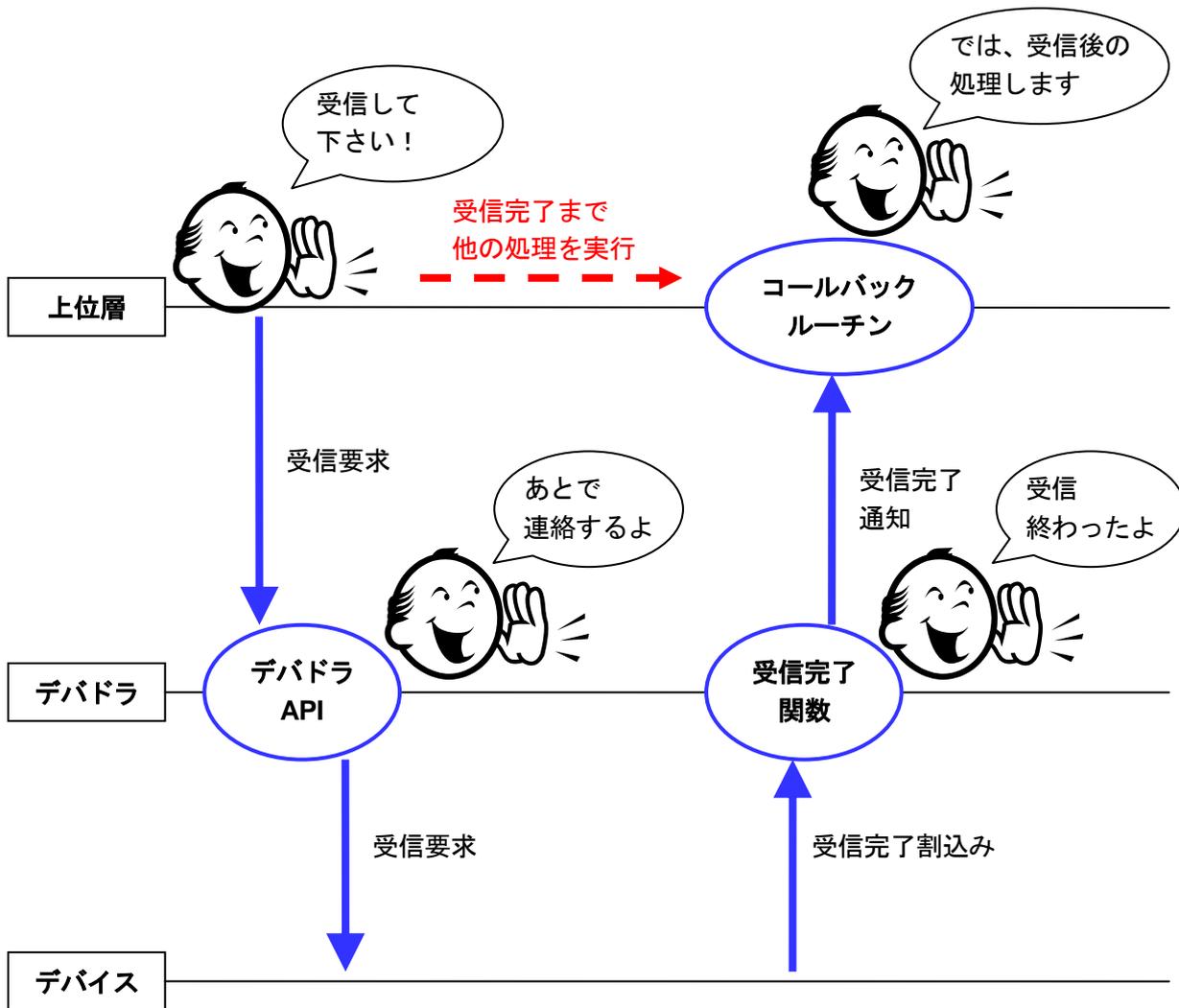



図 9.2-3 コールバック

9.2.4 依存部関数

表 9.2-3は本章で作成する CAN デバイスドライバの依存部関数の一覧です。

表 9.2-3 依存部関数一覧

API 名	関数名	内容
CAN 初期化依存部関数	hw_can_init	CAN コントローラの初期化処理を行います。
CAN 送信設定依存部関数	hw_can_set_trm	メッセージスロット 0 に対して、 送信 ID・送信データ長・送信データを設定し 送信を行います。
CAN 受信設定依存部関数	hw_can_set_rec	指定されたメッセージスロットに対して、 受信 ID を設定し、受信を行います。
CAN 受信完了割込み関数	hw_can_rec_int	CAN メッセージの受信終了時に、 割込みで呼ばれます。 受信完了した ID と受信データを コールバック（下記参照）でアプリに伝えます。 オーバランエラー（データを取り出している 最中に新しいデータを受信するエラー） が発生した場合はそれも通知します。

9.2.4.1　CAN 初期化依存部関数

表 9.2-4は今回作成する CAN 初期化依存部関数の仕様概要、**図 9.2-4**は関数処理のフローチャートです。

表 9.2-4 CAN 初期化依存部関数 hw_can_init 関数仕様

項目			内容
形式			void hw_can_init(void);
処理詳細			CAN0 チャネルに関するレジスタの初期化をします。 入出力ポート、CAN モードレジスタ、CAN 制御レジスタ、CAN ボーレートレジスタ、CAN ボーレートプリスケーラレジスタ、CAN コンフィグレーションレジスタ、CAN グローバルマスクレジスタ 0・1、CAN ローカルマスクレジスタ A0・1、CAN ローカルマスクレジスタ B0・1、CAN 拡張 ID レジスタ、スロットバッファ選択レジスタ、CAN エラー割込みマスクレジスタ、CAN スロット割込みマスクレジスタ、割込み制御レジスタをデバイスドライバの仕様を満たすように初期設定をします。
戻り値	型		void
	コメント		無し
引数	1	型	Void
		名称	
		コメント	無し

図 9.2-4 CAN 初期化依存部関数 hw_can_init フローチャート

9.2.4.2　　CAN 送信設定依存部関数

表 9.2-5は今回作成する CAN 送信設定依存部関数の仕様概要、**図 9.2-5**は関数処理のフローチャートです。

表 9.2-5 CAN 送信設定依存部関数 hw_can_set_trm 関数仕様

項目			内容
形式			UINT8 hw_can_set_trm(UINT16 id, UINT8 dlc, UINT8 *data);
処理詳細			スロットバッファ 0 を通して、メッセージスロット 0 に送信 ID、送信データ長、送信データを設定し、送信要求を出す。 送信データは送信データ長で指定されたデータ数だけ設定する。
戻り値	型		UINT8
	コメント		CAN_E_OK　　　：正常終了 CAN_E_RUNNING ：送信中
引数	1	型	UINT16
		名称	id
		コメント	送信 ID（ 0x0000～0x07FF ）
	2	型	UINT8
		名称	dlc
		コメント	送信データ長（ 0～8 ）
	3	型	UINT8
		名称	*data
		コメント	送信データを格納した配列の先頭ポインタ

図 9.2-5 CAN 送信設定依存部関数 hw_can_set_trm フローチャート

9.2.4.3 CAN 受信設定依存部関数

表 9.2-6は今回作成する CAN 受信設定依存部関数の仕様概要、**図 9.2-6**は関数処理のフローチャートです。

表 9.2-6 CAN 受信設定依存部関数 hw_can_set_rec 関数仕様

項目			内容
形式			UINT8 hw_can_set_rec(UINT16 id, UINT8 slot);
処理詳細			指定されたスロットをスロットバッファ 1 に割り当てる。 スロットバッファ 1 を通して、メッセージスロットに受信 ID を設定し、受信要求を出す。
戻り値	型		UINT8
	コメント		CAN_E_OK ：正常終了 CAN_E_RUNNING ：受信中
引数	1	型	UINT16
		名称	id
		コメント	受信 ID（ 0x0000～0x07FF ）
引数	2	型	UINT8
		名称	slot
		コメント	受信設定するメッセージスロット（ 1～15 ）

図 9.2-6 CAN 受信設定依存部関数 hw_can_set_rec フローチャート

9.2.4.4 CAN 受信完了割込み関数

表 9.2-7は今回作成する CAN 受信完了割込み関数の仕様概要、**図 9.2-7**は関数処理のフローチャートです。

表 9.2-7 CAN 受信完了割込み関数 hw_can_rec_int 関数仕様

<table>
<tr><td colspan="3">項目</td><td>内容</td></tr>
<tr><td colspan="3">形式</td><td>void hw_can_rec_int(void);</td></tr>
<tr><td colspan="3">処理詳細</td><td>CAN 受信割込みによって呼び出される。
受信完了したメッセージスロット番号を取得、メッセージスロットバッファ1 に受信完了したメッセージスロットを割り当てる。
受信完了した標準 ID、受信データ長、受信データを取得、それらを引数としコールバック関数を呼び出す。
また、オーバランエラーを判定し、発生した場合はコールバック関数で通知する。
受信完了フラグ、割込み要求をクリアしておき、次の受信割込みが受け付けられるようにする。</td></tr>
<tr><td rowspan="2">戻り値</td><td colspan="2">型</td><td>void</td></tr>
<tr><td colspan="2">コメント</td><td>無し</td></tr>
<tr><td rowspan="3">引数</td><td rowspan="3">1</td><td>型</td><td>void</td></tr>
<tr><td>名称</td><td></td></tr>
<tr><td>コメント</td><td>無し</td></tr>
</table>

図 9.2-7 CAN 受信完了割込み関数 hw_can_rec_int フローチャート

9.2.5 API

表 9.2-8は本章で作成する CAN デバイスドライバの依存部関数の一覧です。

表 9.2-8 API 一覧

API 名	関数名	内容
CAN 初期化 API	CanInit	CAN 初期化依存部関数を呼出して CAN コントローラの初期化処理を行います。
CAN 送信設定 API	CanSetTrm	引数エラーの判定後、 CAN 送信設定依存部関数を呼出し、 メッセージスロット 0 に対して、 送信 ID・送信データ長・送信データを設定し 送信を行います。
CAN 受信設定 API	CanSetRec	引数エラーの判定後、 CAN 受信設定依存部関数を呼出し、 指定されたメッセージスロットに対して、 受信 ID を設定し、受信を行います。

9.2.5.1　CAN 初期化 API

表 9.2-9は今回作成する CAN 初期化 API の仕様概要、**図 9.2-8**は関数処理のフローチャートです。

表 9.2-9 CAN 初期化 API　CanInit　API 仕様

項目			内容
形式			void CanInit(void);
処理詳細			CAN 初期化依存部関数を呼出して初期化処理を行う。
戻り値	型		void
	コメント		無し
引数	1	型	void
		名称	
		コメント	無し

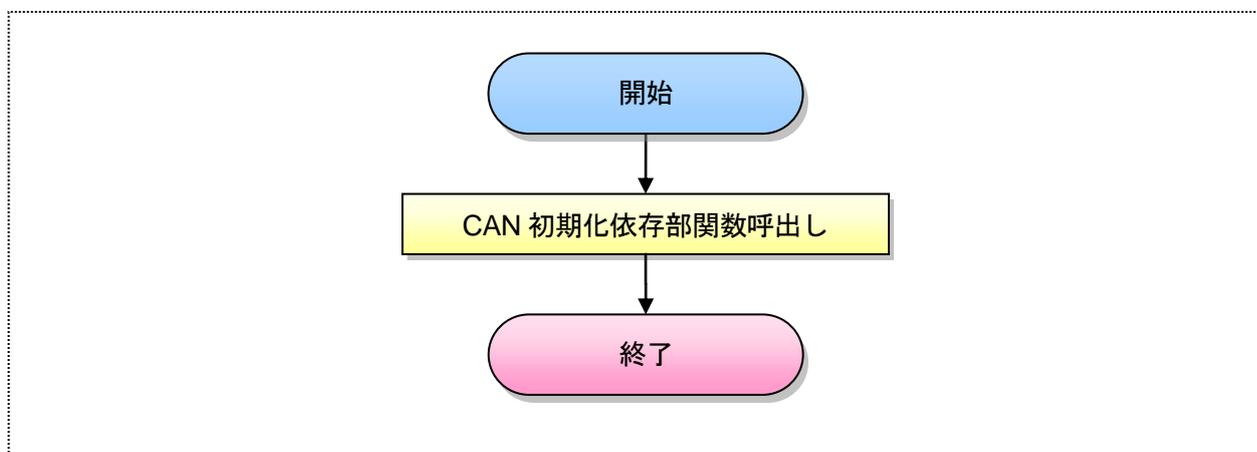



図 9.2-8 CAN 初期化 API　CanInit　フローチャート

9.2.5.2　CAN 送信設定 API

表 9.2-10は今回作成する CAN 送信設定 API の仕様概要、**図 9.2-9**は関数処理のフローチャートです。

表 9.2-10 CAN 送信設定 API　CanSetTrm　API 仕様

項目			内容
形式			UINT8 CanSetTrm(UINT16 id, UINT8 dlc, UINT8 *data);
処理詳細			引数異常を確認後、CAN 送信設定依存部関数を呼出して送信設定処理を行う。
戻り値	型		UINT8
	コメント		CAN_E_OK　：正常終了 CAN_E_PRM　：引数異常 CAN_E_RUNNING　：送信中
引数	1	型	UINT16
		名称	id
		コメント	送信 ID（ 0x0000～0x07FF ）
	2	型	UINT8
		名称	dlc
		コメント	送信データ長（ 0～8 ）
	3	型	UINT8
		名称	*data
		コメント	送信データを格納した配列の先頭ポインタ

図 9.2-9 CAN 送信設定 API　CanSetTrm　フローチャート

9.2.5.3　CAN 受信設定 API

表 9.2-11は今回作成する CAN 受信設定 API の仕様概要、**図 9.2-10**は関数処理のフローチャートです。

表 9.2-11 CAN 受信設定 API　CanSetRec　API 仕様

項目			内容
形式			UINT8 CanSetRec(UINT16 id, UINT8 slot);
処理詳細			引数異常を確認後、CAN 受信設定依存部関数を呼出して受信設定処理を行う。
戻り値	型		UINT8
	コメント		CAN_E_OK　：正常終了 CAN_E_PRM　：引数異常 CAN_E_RUNNING　：受信中
引数	1	型	UINT16
		名称	id
		コメント	受信 ID（ 0x0000～0x07FF ）
引数	2	型	UINT8
		名称	slot
		コメント	受信設定するメッセージスロット（ 1~15 ）

図 9.2-10 CAN 受信設定 API　CanSetRec　フローチャート

9.2.6 コールバック関数

表 9.2-12は受信完了割込み関数で使用するコールバック関数の仕様です。

表 9.2-12 CAN 受信コールバック関数 CanRecCbr 関数仕様

項目			内容
形式			void CanRecCbr(UINT8 rec_err, UINT16 id, UINT8 dlc, UINT8 *data);
処理詳細			CAN 受信割り込みから呼び出される CAN 受信完了を通知するコールバック関数。 引数で正常受信 or オーバランエラー発生、受信完了した ID、受信データ長、受信データを格納した配列の先頭アドレスを渡す。 本関数内での処理内容は上位層に一任する。
戻り値	型		Void
	コメント		無し
引数	1	型	UINT8
		名称	rec_err
		コメント	正常受信 or オーバランエラー発生（CAN_E_OK or CAN_E_OVERRUN）
	2	型	UINT16
		名称	Id
		コメント	受信 ID（ 0x0000～0x07FF ）
	3	型	UINT8
		名称	Dlc
		コメント	受信データ長（ 0～8 ）
	4	型	UINT8
		名称	*data
		コメント	受信データを格納した配列の先頭ポインタ

実際の処理は上位層（デバイスドライバを使用するプログラム。アプリ等）が記述するため、CAN デバイスドライバは can.h にコールバック関数のプロトタイプ宣言を行うだけにします。

上位層は can.h に記述されたプロトタイプ宣言を参照して、上位層のプログラムの中にコールバック関数を記述します。

9.2.7 OIL ファイル（ISR）

CAN デバイスドライバでは CAN 受信割込みを使用して、CAN 受信完了割込み関数を呼出します。そのためには OIL ファイルに ISR オブジェクトを記述する必要があります。**表 5.2-11**は ISR オブジェクトの属性と概要一覧です。

表 9.2-13 ISR オブジェクトの属性と概要一覧

属性	概要
CATEGORY	1：OS 管理外の割込み 2：OS 管理の割込み
RESOURCE	獲得するリソースのリストです
MESSAGE	アクセスするメッセージのリストです。
PRIORITY	ISR の割込み優先度レベルです。（ 1～15 ）
ENTRY	割込み番号です。

CATEGORY は OS 管理外の割込み処理を行うか、OS 管理の割込みを行うかを設定できます。
OS 管理の割込みを使用すると、割込み関数内で OS の API が使用できる、多重割込みが可能、ISR の ID 管理等、様々な処理が行われます。その反面、処理速度は遅くなってしまいます。
OS の機能を使用したいか、処理速度を早くしたいか、割込み処理を発生させる関数の目的・処理内容によって適切な処理を選択する必要があります。
RESOURCE は**6.4 排他制御**で学んだリソースを使用する際に記述します。
リソースは OS の機能ですので、使用する際には CATEGORY を 2：OS 管理の割込みにする必要があります。
MESSAGE は TOPPERS Automotive Kernel の通信ミドルウェアのための属性です。
PRIORITY は割込みが多重発生した際に用いられる優先度レベルです。数字が大きければ大きいほど優先度が高くなります。
CATEGORY1 の最低優先度は CATEGORY2 の最高優先度より高く設定しなければなりません。そうしない場合、システムジェネレータがエラーを出力します。
ENTRY は割込み番号です。割込み番号はハードウエアマニュアルのベクタテーブルを参照して設定します。ここにベクタテーブルに書かれた番号を記述します（M32C/85 ハードウエアマニュアル 11.5.2 可変ベクタテーブル）。

下は UART0 送信完了割込みの記述例です。UartSendIsr は送信完了時に処理させたい関数で、内部でリソース UartResorce を使用することとします。また、他に CATEGORY1、PRIORITY7 の ISR オブジェクトが記述されていることとします。

演習問題

仕様一覧と関数の処理詳細、コールバック関数の仕様を参考に CAN デバイスドライバを作成して下さい。
また、CAN0 受信割込みの設定（ISR オブジェクト）を OIL ファイルに追記して下さい。

9.3 CAN デバイスドライバを用いたアプリケーションの作成

本章では CAN デバイスドライバを用いたアプリケーションを作成します。

9.3.1 システム概要

TOPPERS Platform ボードを 2 枚使用し、 1 枚を送信用、もう 1 枚を受信用とします。
送信側の 4 つのスイッチをそれぞれ押すと､各スイッチごとに異なる送信 ID を持つメッセージを送信します。
受信側は受信に成功した場合、受信した ID とデータを 16 進数で LCD に表示します。

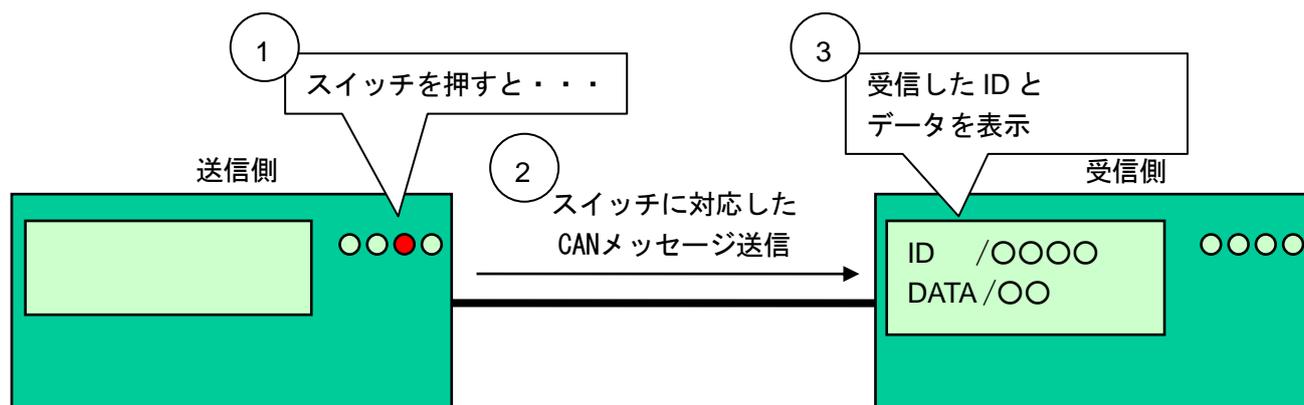



図 9.3-1 CAN 通信アプリケーションシステム概要

9.3.2 ソフトウェア仕様

9.3.2.1 送信アプリケーション

送信アプリケーションはスイッチを監視するタスクと CAN 送信処理を行うタスクから作られます。
また、各デバイスドライバの初期化処理は StartupHook で行われます。

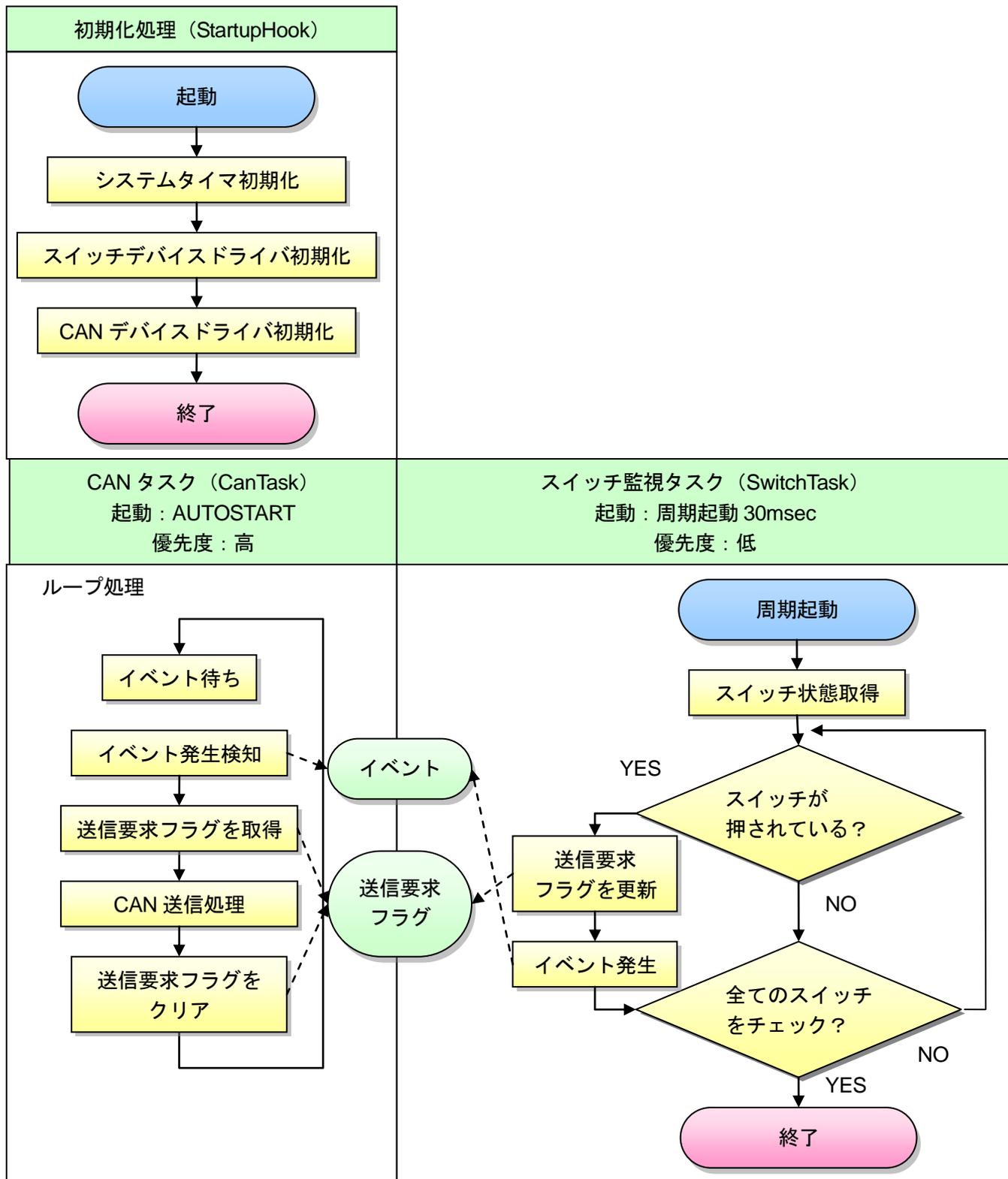



図 9.3-2 送信アプリケーション処理フロー

TOPPERS Platform ボードには計 4 つのスイッチがあり、それぞれに個別の送信 ID・送信データ長・送信データを持ちます。
今回作成する送信アプリケーションでは**表 9.3-1**で示すように送信処理を行います。

表 9.3-1 CAN 送信メッセージ一覧

	送信 ID	送信データ長	送信データ
スイッチ 3	0x0001	1Byte	任意の他と異なる 1Byte のデータ
スイッチ 4	0x0002	1Byte	任意の他と異なる 1Byte のデータ
スイッチ 5	0x0003	1Byte	任意の他と異なる 1Byte のデータ
スイッチ 6	0x0004	1Byte	任意の他と異なる 1Byte のデータ

また、複数のスイッチが同時に押された場合、送信の優先順位は
スイッチ３ ＞ ４ ＞ ５ ＞ ６ とします。
例えば、スイッチ 3 とスイッチ 5 が同時に押された場合、スイッチ 3 による送信処理のみを行います。

演習問題

上記の仕様とフローチャートを参考に CAN 送信アプリケーションを作成して下さい。
また、各タスクを動作させるためのオブジェクトを OIL ファイルに追記して下さい。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex9_tx」フォルダにセットアップ済みです。
①「H25_toppers_ ex9_tx」フォルダをコピー
②「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

演習問題で作成した送信アプリケーションを動作させて、スイッチ３を押したときの CAN0out 端子から出力される信号波形をオシロスコープで観測してデータフレームを解析しましょう。
④ OSF から ITM まで 1 ビット単位でドミナント（0）／リセッシブ（1）を記録しましょう。
⑤ Id（11 ビット）はいくつになっていますか。
⑥ スタッフィングルールが適用されているのがわかりますか。
⑦ ID,RTR,DLC,データフィールド、CRC,ACK,EOF,ITM を確認しましょう。

9.3.2.2　受信アプリケーション

受信アプリケーションは LCD・LED を表示するタスクと CAN 受信割込み完了によって呼び出されるコールバック関数から作られます。

各デバイスドライバの初期化処理及び、CAN 受信設定は StartupHook で行われます。

LCD の表示更新である 100msec 中に複数のメッセージが受信された時は、最後に受信したメッセージを表示します。また、LCD の初期表示は CAN 受信が行われるまで続けます。

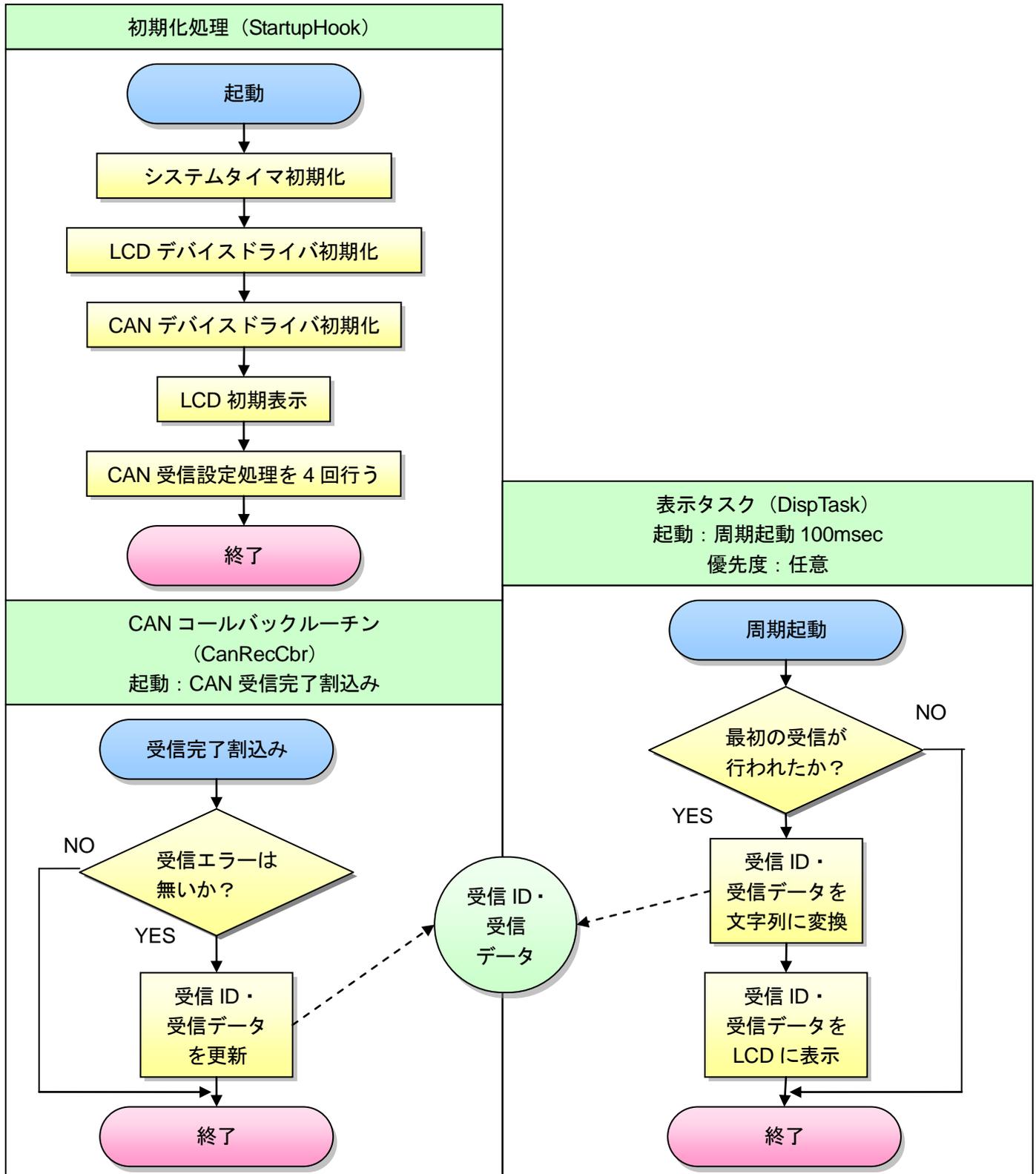



図 9.3-3 受信アプリケーション処理フロー

CAN 受信設定は送信側の送信 ID に合わせ、**表 9.3-2**のように 4 回実行します。

表 9.3-2 CAN 受信 ID 一覧

メッセージスロット番号	受信 ID
1	0x0001
2	0x0002
3	0x0003
4	0x0004

LCD には ID・受信データを 16 進数で**図 9.3-4**のように表示します。

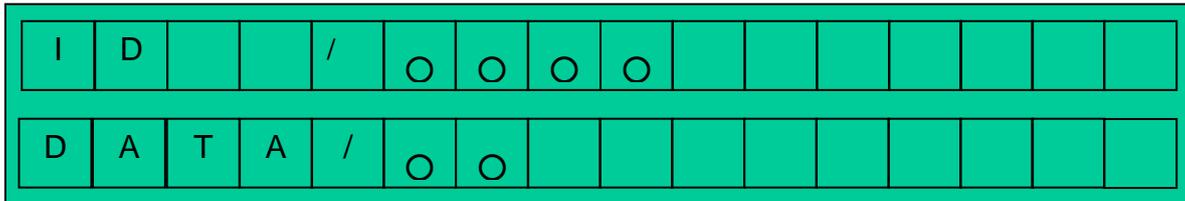



図 9.3-4 LCD 表示

1 列目に受信 ID、2 列目に受信データを表示し、最初の 5 文字を見出しとします。実際のデータは 6 文字目から表示します。
受信 ID の範囲は 16 進数で 0x0000～0x07FF なので、LCD には 16 進数 4 文字で表示します。
CAN の通信データは 1Byte なので、LCD には 16 進数 2 文字で表示します。

LCD は文字列しか表示できませんが、受信する ID・データは数値である為、数値から文字列の変換処理が必要となります。
数値から文字列変換の API がシリアルドライバに含まれているので、それらを使用しましょう。

表 9.3-3 16bit 数値 -> 16 進数文字列変換関数 ConvShort2HexStr 関数仕様

項目			内容
形式			void ConvShort2HexStr(UINT8 *dst, UINT16 src)
処理詳細			引数 src の数値を文字列に変換し、引数*dst に格納します。 また、終端文字’¥0’が 5 文字目に付けられます。 dst は 5Byte 以上（4 桁＋終端文字）以上保持することを前提とします。
戻り値	型		Void
	コメント		戻り値なし
引数	1	型	UINT8
		名称	*dst
		コメント	変換した文字列を格納するバッファの先頭アドレス
引数	2	型	UINT16
		名称	src
		コメント	変換する数値

表 9.3-4 8bit 数値 -> 16 進数文字列変換関数 ConvByte2HexStr 関数仕様

項目			内容
形式			void ConvByte2HexStr(UINT8 *dst, UINT8 src)
処理詳細			引数 src の数値を文字列に変換し、引数*dst に格納します。 また、終端文字'¥0'が 3 文字目に付けられます。 dst は 3Byte 以上（2 桁＋終端文字）以上保持することを前提とします。
戻り値	型		Void
	コメント		戻り値なし
引数	1	型	UINT8
		名称	*dst
		コメント	変換した文字列を格納するバッファの先頭アドレス
引数	2	型	UINT8
		名称	src
		コメント	変換する数値

演習問題

上記の仕様とフローチャートを参考に CAN 受信アプリケーションを作成して下さい。
また、各タスクを動作させるためのオブジェクトを OIL ファイルに追記して下さい。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_ex9_rx」フォルダにセットアップ済みです。
①「H25_toppers_ ex9_rx」フォルダをコピー
②「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示

9.3.3 TOPPERS Platform ボード間の接続方法

TOPPERS Platform ボードで CAN 通信を行うには、SW7 のスイッチを ON にします。

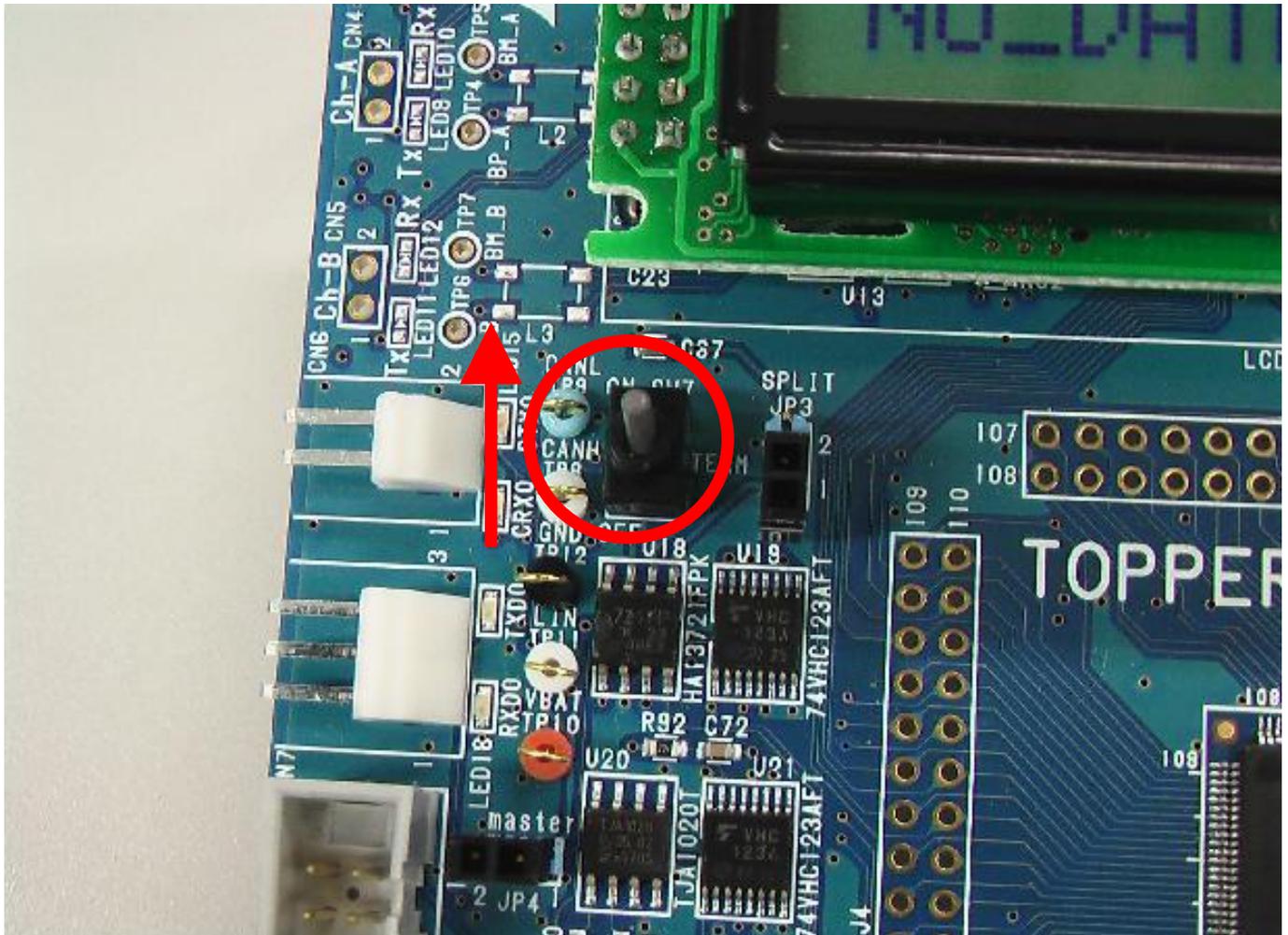



図 9.3-5 CAN 通信スイッチ設定

CAN ケーブルで TOPPERS Platform ボード間を接続します。

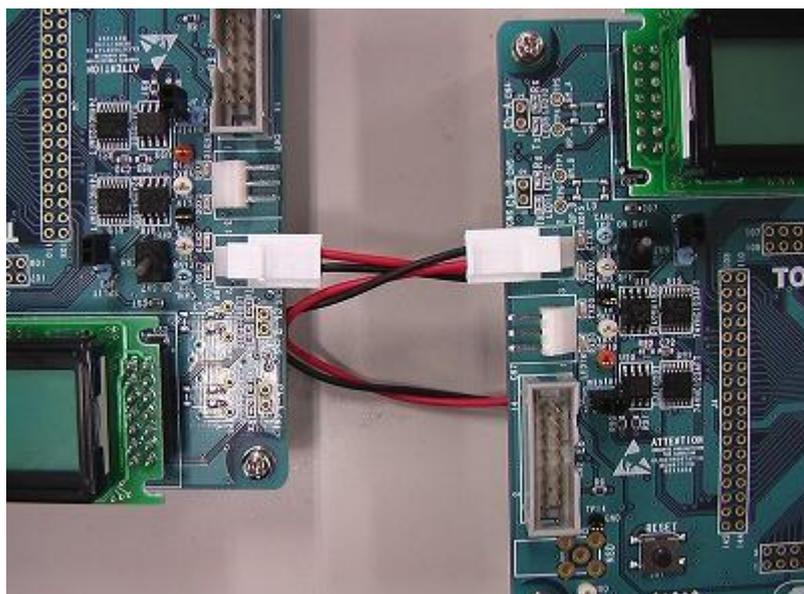



図 9.3-6 CAN ケーブル接続

では、アプリを動かしてみましょう。
受信側の LCD の初期状態は下図のようになります。

図 9.3-7 CAN 受信アプリ LCD 初期状態

送信側のボタンを押すと受信側の LCD に受信されたメッセージの ID とデータが表示されます。
ボタンに対応した ID と、設定したデータが表示されるか確認しましょう。
もし、ID の設定が送信アプリと受信アプリとで違っている場合、CAN 通信は失敗となり、LCD に表示されません。

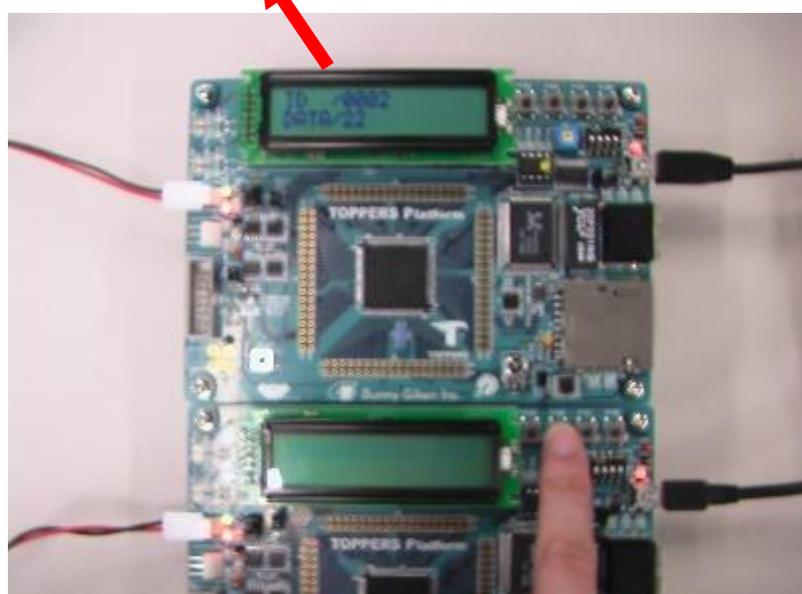



図 9.3-8 CAN 通信アプリ成功

図 9.3-9は CAN 通信成功時の LED、**図 9.3-10**は CAN 通信失敗時の LED となります。
図の右側の TOPPERS Platform ボードが送信、左側が受信となります。

CAN 通信成功時は送信・受信共、LED が点灯します

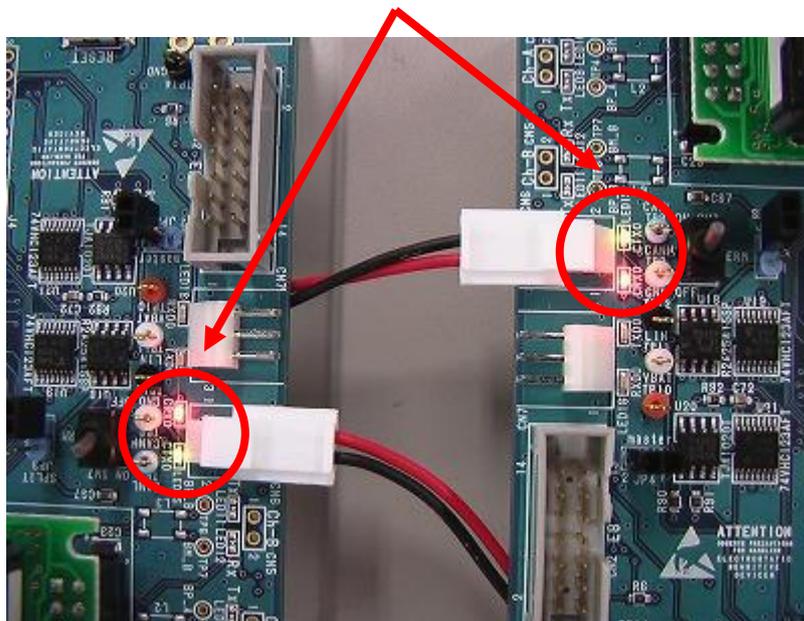



図 9.3-9 CAN 通信成功時の LED

CAN 通信失敗時は受信側の LED が点灯しません

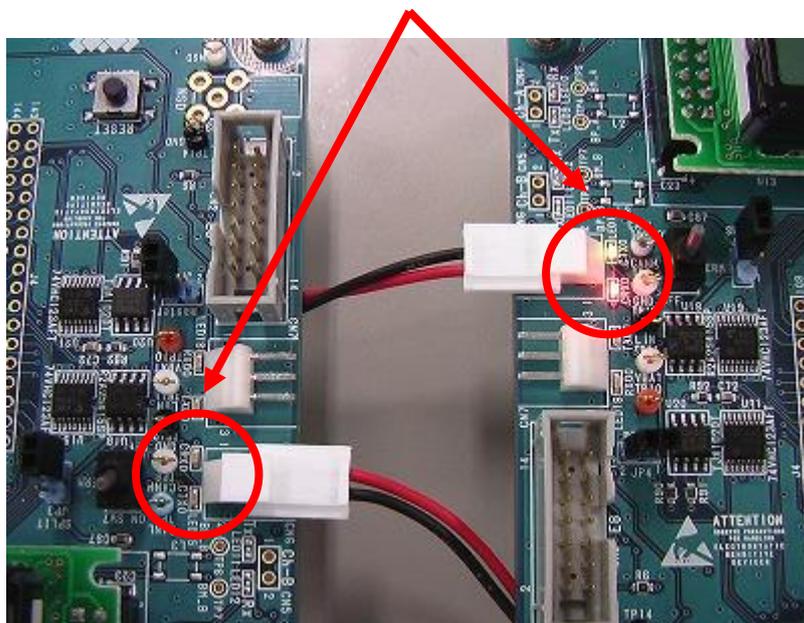



図 9.3-10 CAN 通信失敗時の LED

10 応用アプリケーションの開発

10.1 概要

本章では、ＣＡＮコントローラ（ルネサス製マイコン　M32C/85 グループの M30855FJGP ）、ＣＡＮ通信プロトコル、OBD-Ⅱ（On-board diagnostics-Ⅱ）マルチタスクプログラミング、リアルタイムＯＳを活用したシステム事例によって理解を深め、自動車組込み技術の構成要素を学習することを目的として、ＣＡＮコントローラによる簡易距離計測システムの開発演習を行います。

10.1.1 システム概要

図 10.1-1 に簡易距離計測システム全体のシステム構成図を示します。ＦＰＧＡボードとマイコンボードの通信はＣＡＮバスを介して行います。

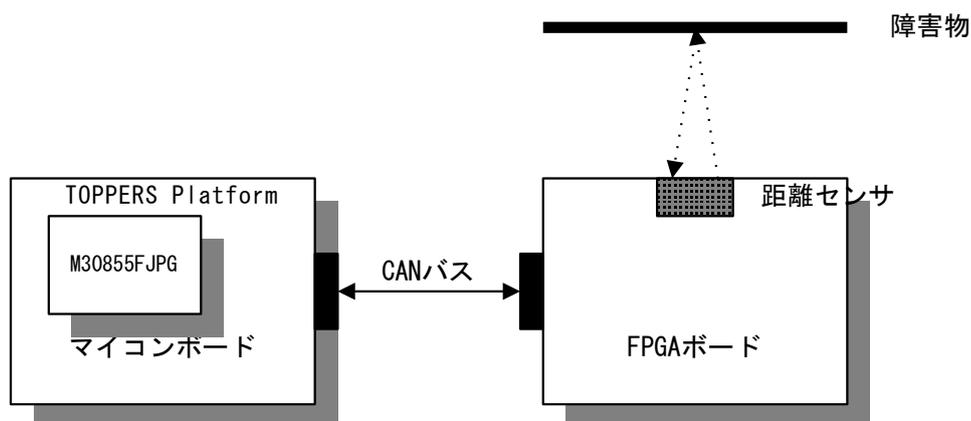
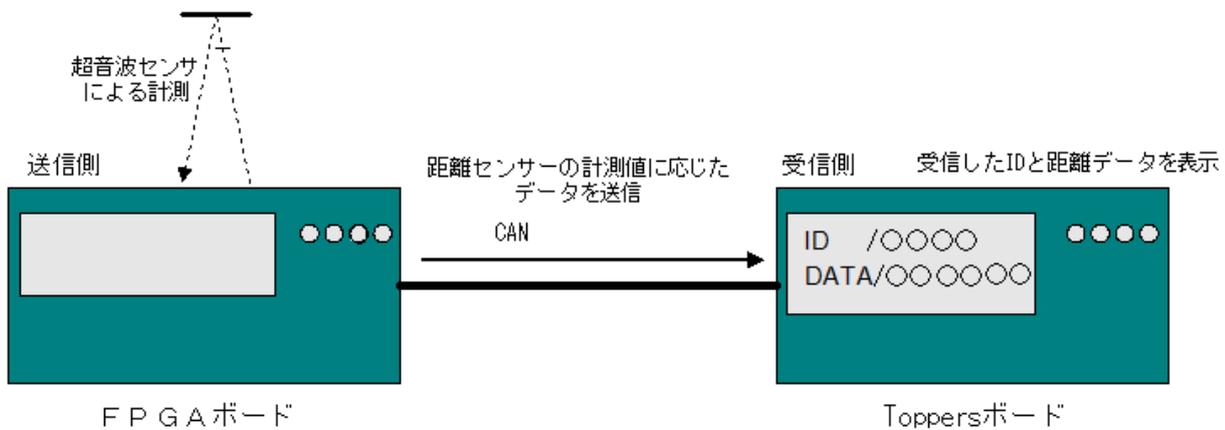



図 10.1-1 簡易距離計測システムの構成

注意：ＦＰＧＡボードの詳細は、テキスト「　　　　　　　　」を参照

10.1.2　　システム仕様

ＦＰＧＡボードより送信される距離計測データをマイコンボードの液晶に表示します。

10.1.3　　CAN 通信仕様

①　通信プロトコル

- 転送速度　　　　　　：５００Kbps
- メッセージフレーム　：データフレーム（ID は 29 ビット）
- フォーマット　　　　：　2.0B アクティブ（ID は 29 ビット拡張フォーマット）
- 送信 ID　　　　　　　：なし
- 受信 ID　　　　　　　：0x00040000（base id = 1）⇒距離データ　（図 10.1-2 参照）
    - コントロールフィールド：DLC（Data Length Code）= 8

②　実際の通信データ

距離センサからの距離データの受信：FPGA ボードからのデータフレーム構成（図 10.1-2 参照）

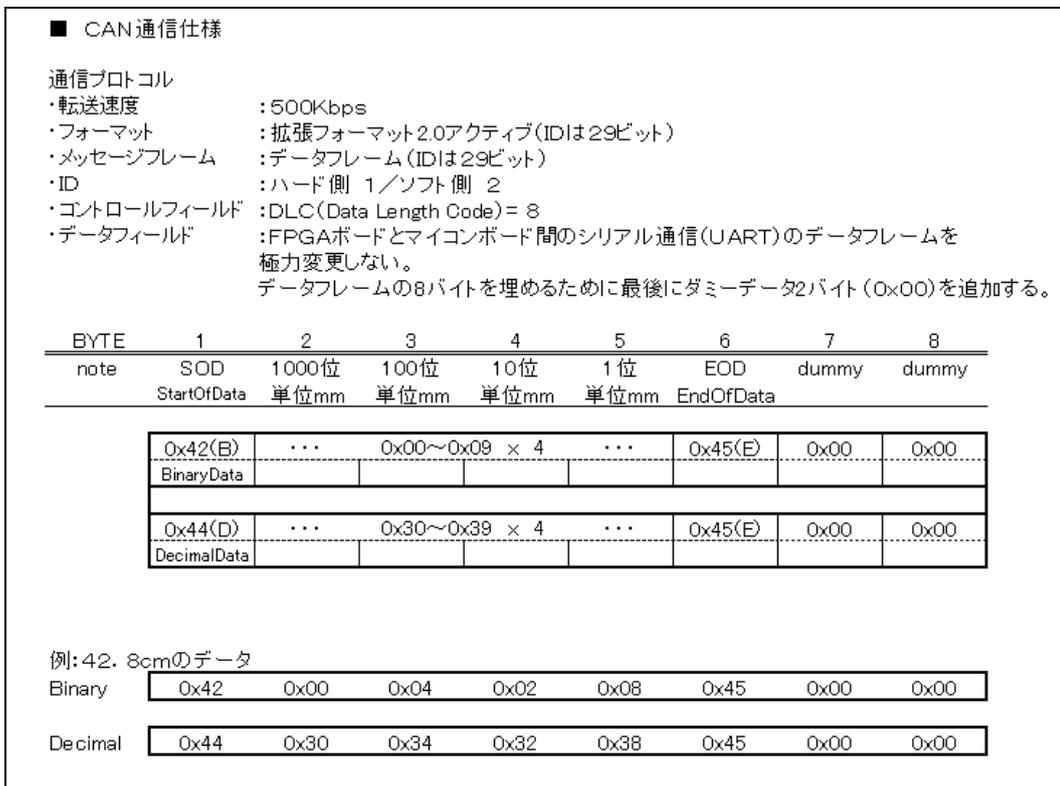



図 10.1-2 距離センサのデータフレーム構成

10.2 ハードウェア環境

10.2.1 TOPPERS Automotive Kernel 実装環境

ターゲットボード（TOPPERS Platform）のハードウェア一般仕様及び通信仕様を下記に示します。

表 10.2-1 TOPPERS Platform ボードのハードウェア一般仕様

No.	項目	仕様
1	製品名	TOPPERS Platform ボード
2	製品型番	S810-TPF-85
3	搭載 MCU	M32C/85グループ：M30855FJGP 内蔵RAM ：24K ROM ：512K SFR 領域：１K
4	メインクロック	8MHz（XIN）
5	SRAM	128KByte×２ （M5M51008DVP-55H ルネサス）
6	電源電圧	5V（USB より供給）
7	消費電力	最大 約100mA（メモリーカード非動作時）
8	外形寸法	約 150mm（W） × 110mm（D） × 45mm（H）
9	重量	約 120g

表 10.2-2 TOPPERS Platform ボードのハードウェア通信仕様

No.	項目	仕様
1	CAN	×１ch トランシーバ：HA13721FPK（ルネサス）
2	LIN	×1ch トランシーバ：TJA1020（Philips）

3	Ethernet	10BASET×１ コントローラ：RTL8019AS（RealTek）
4	SD/MMC カード	スロット×１（SPI モードによるアクセス）
5	USB	×１（Full Speed 対応） ドライバ：FT232R（FTDI）

表 9.2-2は、ターゲットとなるマイコン M32C/85 が持つ CAN コントローラの機能の仕様と、今回作成するデバイスドライバで使用する機能及び設定の一覧です。

表 10.2-3 CAN コントローラ機能及びデバイスドライバ実装機能・設定一覧

CAN コントローラ搭載機能	デバイスドライバ実装機能及び設定
CAN チャネルを 2 つ搭載	CAN0 チャネルを使用します。 （本ボードは CAN0 チャネルのみ端子が搭載されています。） CAN0 出力ポートは P7_6、CAN0 入力ポートは P7_7 を使用します。
メッセージスロットを 16 本（0～15）搭載	メッセージスロット 0 を送信用、 メッセージスロット 1～15 を受信用として使用します。
メッセージスロットバッファを 2 つ搭載 （メッセージスロットにアクセスするときは、 このバッファを通してアクセスします。）	バッファ 0 にはメッセージスロット 0 を、 バッファ 1 にはメッセージスロット 1～15 を割り当てます。
標準 ID か拡張 ID、どちらかを選択できます。	今回の演習では拡張 ID を使用します。
アクセプタンスフィルタ マスクレジスタでフィルタの設定ができます。 ・ グローバルマスクレジスタ ･･･メッセージスロット 0～13 用 ・ ローカルマスク A ･･･メッセージスロット 14 用 ・ ローカルマスク B ･･･メッセージスロット 15 用	アクセプタンスフィルタを使用します。
転送速度 最大転送速度 1Mbps	転送速度を 500Kbps に設定します。 ビットタイミングは、 PTS ：5Tq PBS1：6Tq PBS2：4Tq SJW ：3Tq サンプリング回数：3 回 と設定します。
エラー検出 ビットエラー、ACK エラー、CRC エラー、 スタッフエラー、フォーマットエラーの検出が 可能です	エラーが発生しても、その通知を行うことはしません。
割込み 送信終了時、受信終了時、エラー発生時に 割込みを発生させることが出来ます。	受信終了の割込みを使用します。
タイムスタンプ機能	使用しませんが、タイムスタンププリスケーラは 念のため CAN バスビットクロックで初期化設定しておきます。
リモートフレーム自動応答機能	使用しません。
送信アボート機能	使用しません。
ループバック機能	使用しません。
エラーアクティブ強制復帰機能	使用しません。
シングルショット送信機能	使用しません。
自己診断機能	使用しません。

10.2.2　搭載MCUのCANモジュール仕様

ルネサス製マイコン　M32C/85 グループは､CAN2.0B 仕様準拠の Full CAN モジュールを 2 チャンネル（CAN0､CAN1）内蔵しています。今回は CAN0 を使用します。表 10.2-4 に CAN モジュールの仕様、図 10.2-1 に CAN モジュールのブロック図、図 10.2-2 にメッセージスロットとメッセージスロットバッファの関係を示します。

図の中で i は CAN モジュールのチャンネル番号とメッセージスロットバッファ番号（i=0～１）を、j はメッセージスロット番号（j=0～15）です。

表 **10.2-4** CAN モジュールの仕様

メッセージスロット数	16本
極性	ドミナント：“L” レセシブ　：“H”
アクセプタンスフィルタ	グローバルマスク：1本（メッセージスロット0～13に対応） ローカルマスク　：2本（それぞれメッセージスロット14、15に対応）
転送速度	転送速度 = $\dfrac{1}{\text{Tq 周期} \times \text{1ビット分のTqの数}}$ … Max 1 Mbps Tq周期 = $\dfrac{\text{BRP} + 1}{\text{CANクロック}}$ 1ビット分のTqの数 = SS ＋ PTS + PBS1 + PBS2 Tq　: Time quantum BRP　: C0BRP、C1BRPレジスタの設定値。1～255 SS　: Synchronization Segment。1 Tq PTS　: Propagation Time Segment。1～8Tq PBS1 : Phase Buffer Segment 1。2～8Tq PBS2 : Phase Buffer Segment 2。2～8Tq
リモートフレーム自動応答機能	リモートフレームを受信したメッセージスロットが自動的にデータフレームの送信を行う機能
タイムスタンプ機能	16ビットカウンタによるタイムスタンプ機能。カウントソースはCANバスビットクロックの1、2、3、4分周を選択可能 CANバスビットクロック = $\dfrac{1}{\text{CANビットタイム}}$
BasicCANモード	メッセージスロット14、15を使用してBasicCAN機能を実現
送信アボート機能	送信要求を取り消す機能
ループバック機能	CANモジュールが送信したフレームを同CANモジュールが受信する機能
エラーアクティブ強制復帰機能	エラーカウンタをリセットすることにより、強制的にエラーアクティブ状態に遷移させる機能
シングルショット送信機能	アービトレーションロストや送信エラー発生により送信に失敗しても再送信しない機能
自己診断機能	CANモジュール内部で通信を行い、通信モジュールを診断する機能

注1.　発振最大許容誤差1.58%を満たす発振子をご使用ください。

図 10.2-1 から、内部バスはメッセージスロットバッファｉには接続されていますがメッセージスロットｊには直接つながっていないことがわかります。つまり、メッセージスロットｉはＣＰＵから直接アクセスできません。アクセスする場合は、使用するメッセージスロットｊを CANi メッセージスロットバッファに割り当て、この番地を通してアクセスします。

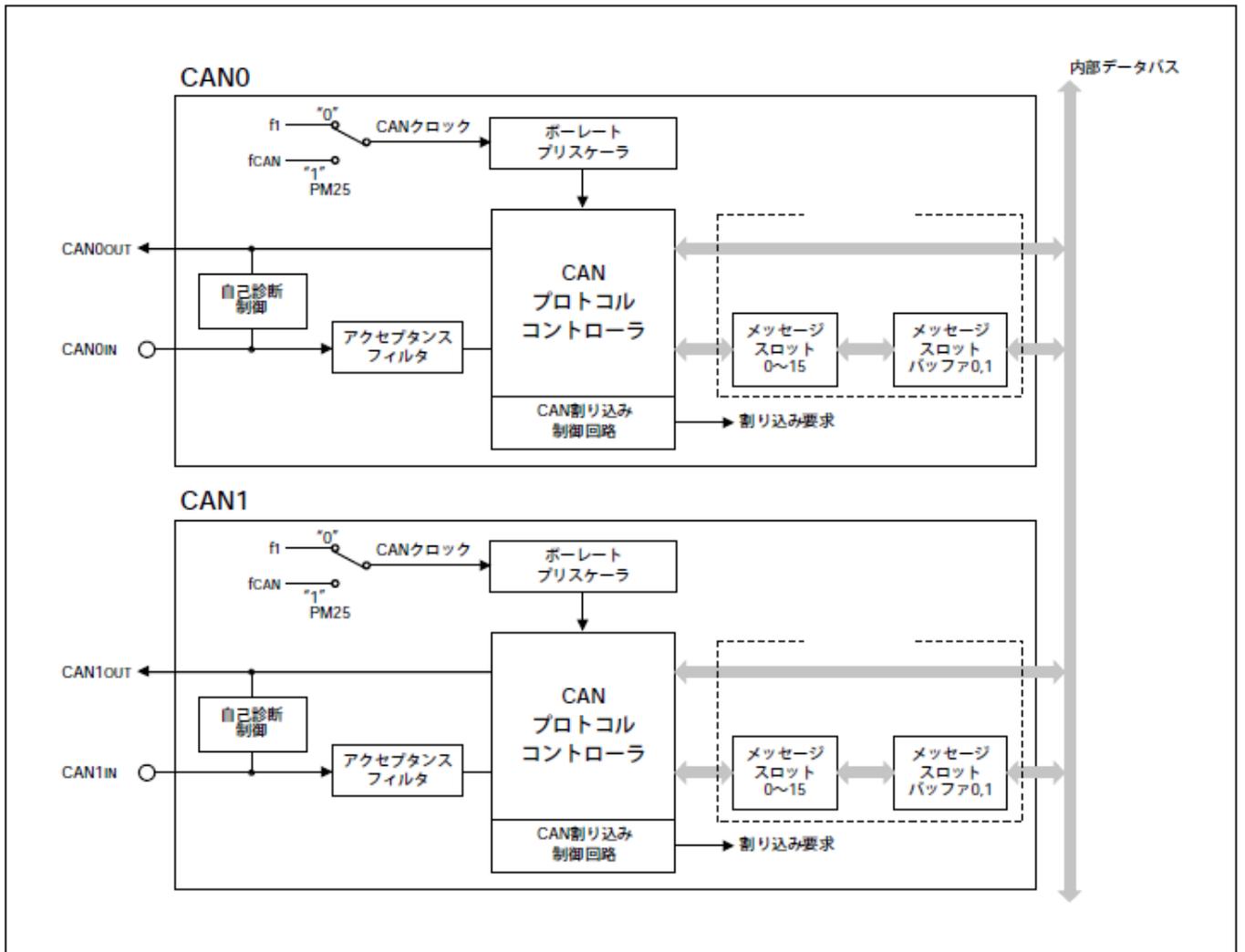



図 **10.2-1** CAN モジュールのブロック図

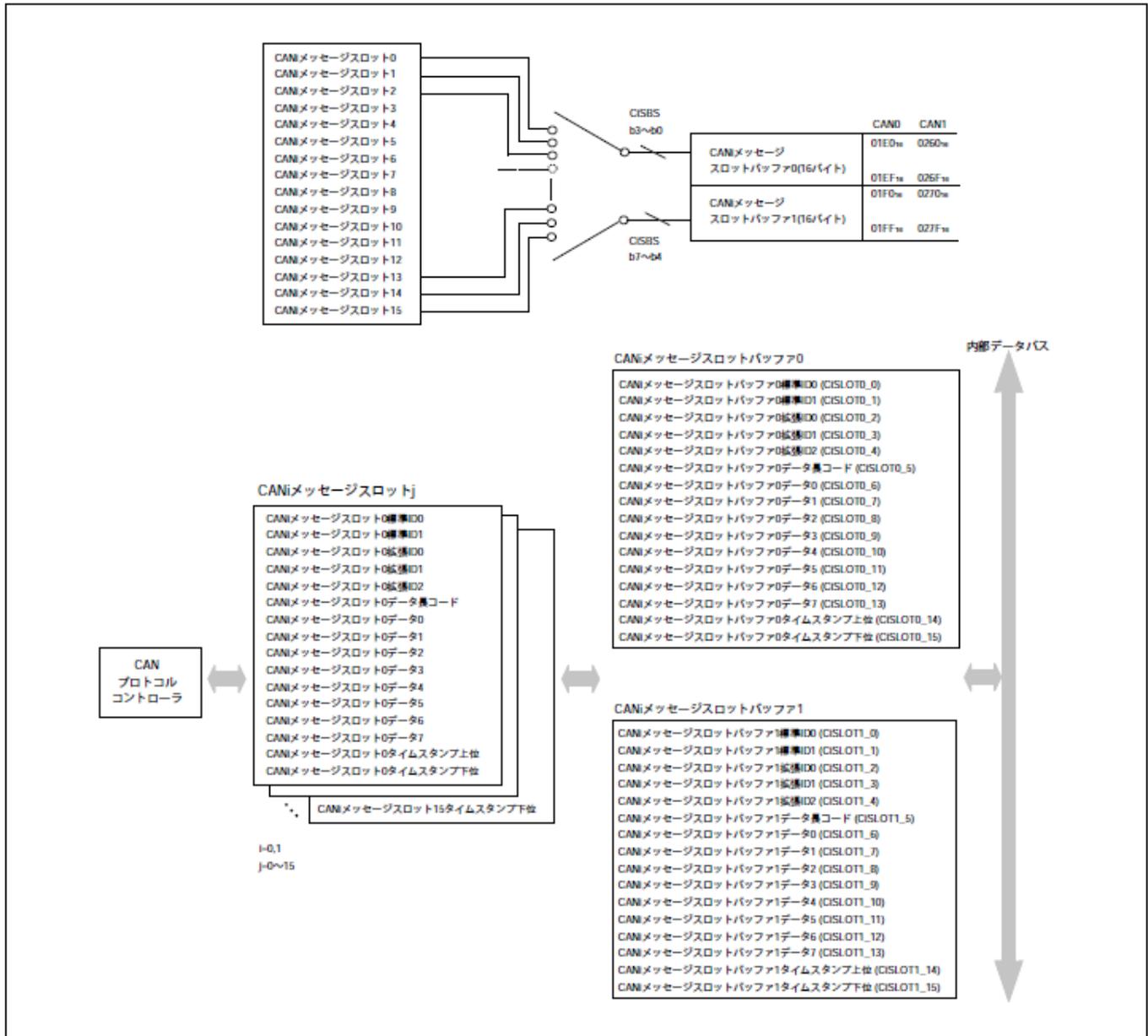



図 **10.2-2** メッセージスロットとメッセージバッファ

CANi メッセージスロットバッファ 0、1 に割り当てるメッセージスロット j の選択は CANi スロットバッファ選択レジスタ（CiSBS）で行います。

今回のメッセージスロットおよびメッセージスロットバッファの使用方法を図 10.2-3 に示します。

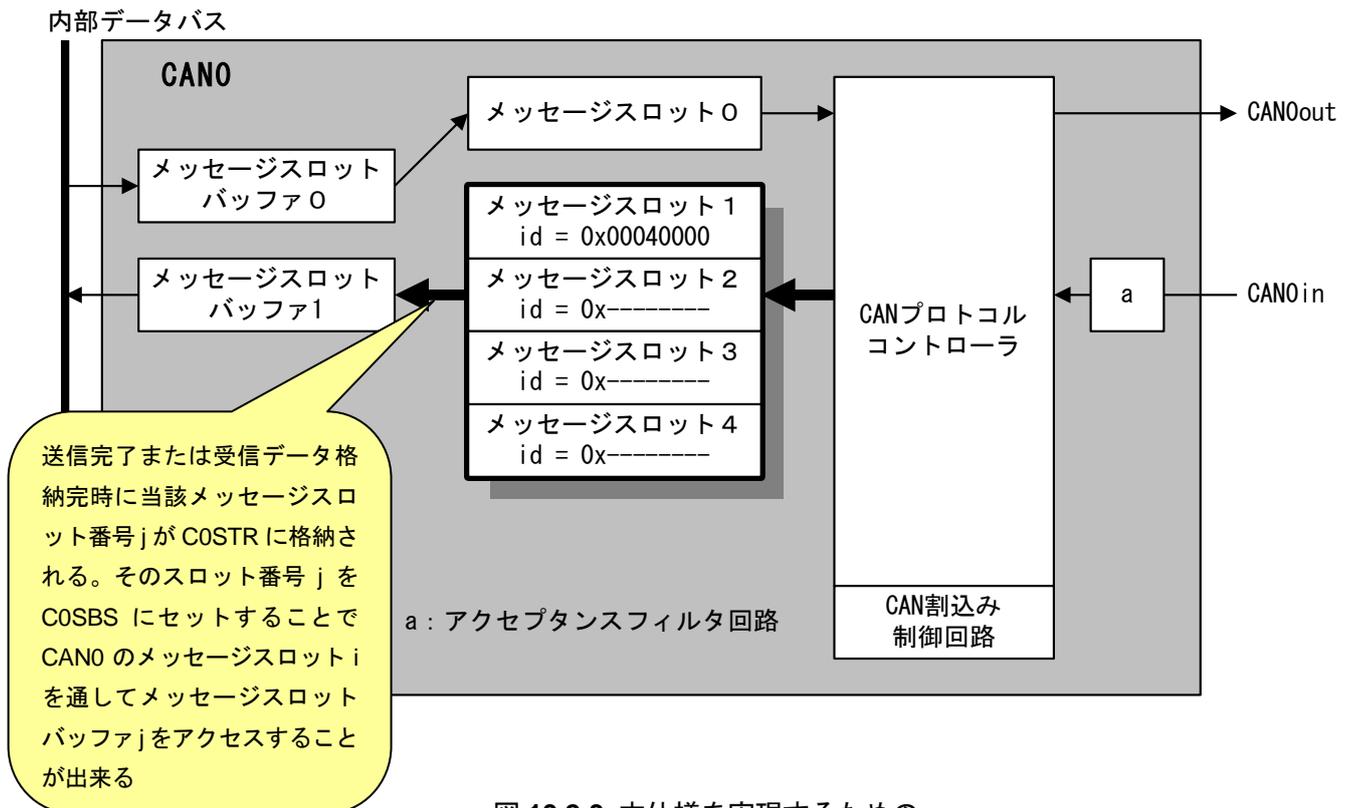



図 **10.2-3** 本仕様を実現するための
メッセージスロットおよびメッセージスロットバッファの割付

送信：メッセージスロットバッファ０をメッセージスロット０に割り付けて送信します。
受信：メッセージスロットバッファ 1～４に受信するＩＤ（拡張ＩＤ29 ビット）を割り付けて、メッセージスロットバッファ１を使って受信します。簡易距離計測システムではメッセージスロット１のみ使用します。

10.3 簡易距離計測システムの動作

ＦＰＧＡボードとＴｏｐｐｅｒｓボードをｃａｎケーブルで接続します。
Toppers ボードのソフトウェアは下記手順で行います。

簡単操作

新規プロジェクトの作成からアプリケーションの作成までの一連操作が「h25_toppers_platfoem_v1.00」フォルダにセットアップ済みです。

① 「H25_toppers_ platfoem_v1.00」フォルダをコピー
② 「SAMPLE.hws」をダブルクリック
③ ＨＥＷの画面が表示
④ 全てをビルド→ダウンロード→実行

10.4 OBD―Ⅱ

OBD−Ⅱとは On−board diagnostics second generation の略で、それまで各社ばらばらだったアダプタ形状、故障コード等を規格統一したもので、自動車に搭載されるECU(Erectric　Control Unit)の自己診断のことです。専用のケーブルで｢OBDII コネクター｣に接続し、エンジン回転数、水温、などを表示することができます。
写真 10−1 はOBDケーブルを自動車本体と Toppers Platform ボード上のCANコネクタに接続できるようにOBDケーブルの片側を改修した例です。。

写真 10-1 ＯＢＤケーブルのアダプタ形状（オス側）とＣＡＮコネクタ

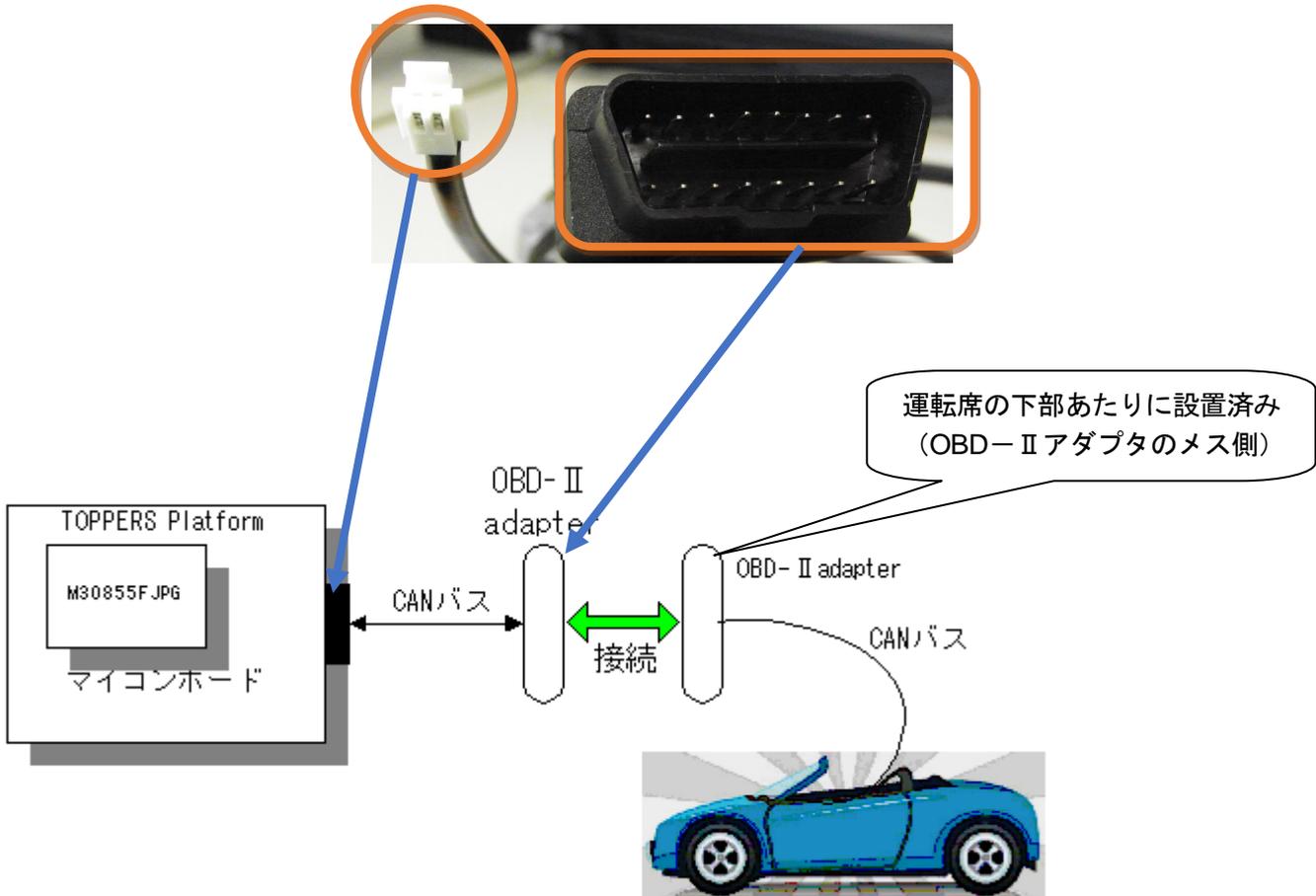



図 10.4-1 実車の各種データの読み出しの概要

エンジン回転数、水温などを知りたいときは、各ＥＣＵのＩＤを設定したデータフレームを送信すると、当該データがデータフレームで返送されてきます。ＥＣＵに対してデータ返送を要求するときとその要求に応じて各ＥＣＵが返送してくるデータフレームの構成を説明します。

■ＥＣＵに対してデータの返送を要求するデータフレーム
ＩＤ　　：２９ビット（種々の事情を勘案して明記しません。メーカー、車種、ＥＣＵによって異なります）
ＤＬＣ　：0x08（常に８バイトのデータが続く：data1～data8）
Data1　：0x02（次に続くデータバイト数）
Data2　：Mode（表 10.4-1 参照）
Data3　：PID（表 10.4-2 参照）
Data4～data8：0x00（ダミーデータ）

■要求に応じて各ＥＣＵが返送してくるデータフレーム

ＩＤ　　：２９ビット（種々の事情を勘案して明記しません。メーカー、車種、ＥＣＵによって異なります）
ＤＬＣ　：0x08（常に８バイトのデータが続く：data1～data8）
Data1　：0x＊＊（次に続くデータバイト数：PID によって変化）
Data2　：Mode+0x40（表 10.4-1 参照）
Data3　：PID（表 10.4-2 参照）
Data4　：データ
Data5　：データ
Data6　：データ
Data7　：データ
Data8　：データ

Mode (hex)	Description
01	Show current data
02	Show freeze frame data
03	Show stored Diagnostic Trouble Codes
04	Clear Diagnostic Trouble Codes and stored values
05	Test results, oxygen sensor monitoring (non CAN only)
06	Test results, other component/system monitoring (Test results, oxygen sensor monitoring for CAN only)
07	Show pending Diagnostic Trouble Codes (detected during current or last driving cycle)
08	Control operation of on-board component/system
09	Request vehicle information
0A	Permanent Diagnostic Trouble Codes (DTCs) (Cleared DTCs)

表 10.4-1　OBD－Ⅱ　Mode 一覧

PID (hex)	Data bytes returned	Description	Min value	Max value	Units	Formula[a]
00	4	PIDs supported [01 - 20]				Bit encoded [A7..D0] == [PID $01..PID $20] See below
01	4	Monitor status since DTCs cleared. (Includes malfunction indicator lamp (MIL) status and number of DTCs.)				Bit encoded. See below
02	2	Freeze DTC				
03	2	Fuel system status				Bit encoded. See below
04	1	Calculated engine load value	0	100	%	A*100/255
05	1	Engine coolant temperature	-40	215	°C	A-40
06	1	Short term fuel % trim—Bank 1	-100 Subtracting Fuel (Rich Condition)	99.22 Adding Fuel (Lean Condition)	%	(A-128) * 100/128
07	1	Long term fuel % trim—Bank 1	-100 Subtracting Fuel (Rich Condition)	99.22 Adding Fuel (Lean Condition)	%	(A-128) * 100/128
08	1	Short term fuel % trim—Bank 2	-100 Subtracting Fuel (Rich Condition)	99.22 Adding Fuel (Lean Condition)	%	(A-128) * 100/128
09	1	Long term fuel % trim—Bank 2	-100 Subtracting Fuel (Rich Condition)	99.22 Adding Fuel (Lean Condition)	%	(A-128) * 100/128
0A	1	Fuel pressure	0	765	kPa (gauge)	A*3
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0C	2	Engine RPM	0	16,383.75	rpm	((A*256)+B)/4
0D	1	Vehicle speed	0	255	km/h	A
0E	1	Timing advance	-64	63.5	° relative to #1 [illegible]	(A-128)/2

表 10.4-2　OBD－Ⅱ　PID 一覧の抜粋

例としてエンジン回転数を調べるときは
表 10.4-1 より Mode=0x01（Show current data)、表 10.4-2 より PID=0x0C（Engine RPM）として対象ＥＣＵのＩＤを設定したデータフレームを送信します。
エンジン回転数を読み出すときのキャナライザーで見たＣＡＮデータフレームの信号波形およびその解析データを図 10.4-2 に示します。

図 10.4-2　エンジン回転数読み出し時のデータフレームおよび解析データ

11 演習問題解答例

11.1 6.3 章 - アラーム

11.1.1 OIL ファイル

【sample.oil】

```
#include "implementation.oil"

CPU current {
#include <t100us_timer.oil>

        OS os {
                STATUS = STANDARD;
                STARTUPHOOK = TRUE;
                ERRORHOOK = FALSE;
                SHUTDOWNHOOK = FALSE;
                PRETASKHOOK = FALSE;
                POSTTASKHOOK = FALSE;
                USEGETSERVICEID = TRUE;
                USEPARAMETERACCESS = TRUE;
                USERESSCHEDULER = FALSE;
        };

        APPMODE AppMode1 {};

        TASK LedOnTask {
                AUTOSTART = FALSE;
                PRIORITY = 14;
                STACKSIZE = 0x0100;
                ACTIVATION = 1;
                SCHEDULE = FULL;
        };

        TASK LedOffTask {
                AUTOSTART = FALSE;
                PRIORITY = 14;
                STACKSIZE = 0x0100;
                ACTIVATION = 1;
                SCHEDULE = FULL;
        };

        ALARM LedOnAlm{
                COUNTER = T100usTimerCnt;     /* 1ms 毎に 1 カウントアップするカウンタ */
                ACTION  = ACTIVATETASK { TASK = "LedOnTask"; };
                AUTOSTART = TRUE {
                                                APPMODE = AppMode1;
                                                ALARMTIME = 3000;
                                                CYCLETIME = 6000;
```

```
                                        };
        };

        ALARM LedOffAlm{
                COUNTER = T100usTimerCnt;
                ACTION  = ACTIVATETASK { TASK = "LedOffTask"; };
                AUTOSTART = FALSE;
        };
};
```

11.1.2　ヘッダファイル

【sample.h】

```
/*****************************************************************************/
/*   インクルードファイル                                                    */
/*****************************************************************************/
#include "kernel_id.h"

/*****************************************************************************/
/*   定数定義                                                                */
/*****************************************************************************/

/*****************************************************************************/
/*   外部宣言                                                                */
/*****************************************************************************/
DeclareAlarm( LedOffAlm );
DeclareTask( LedOnTask );
DeclareTask( LedOffTask );
```

11.1.3　ソースファイル

【sample.c】

```
/*****************************************************************************/
/*   インクルードファイル                                                    */
/*****************************************************************************/
#include "kernel.h"
#include "t100us_timer.h"
#include "sample.h"
#include "led.h"

/*****************************************************************************/
/*   定数定義                                                                */
/*****************************************************************************/
#define ALARM_CNT ((UINT32)3000ul)             /* 300ms(1カウント:100μs) */


/*****************************************************************************/
/*   外部宣言                                                                */
/*****************************************************************************/

/*****************************************************************************/
/*   内部宣言                                                                */
/*****************************************************************************/
void main( void );
TASK( LedOnTask );
TASK( LedOffTask );

/*****************************************************************************/
```

```
/* 関数名 : void main( void )                                              */
/* 役割   : メイン処理                                                       */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/****************************************************************************/
void main( void )
{
        StartOS( AppMode1 );                /* OS スタート */
}

/****************************************************************************/
/* 関数名 : TASK( LedOnTask )                                               */
/* 役割   : LED 点灯                                                         */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/****************************************************************************/
TASK( LedOnTask )
{
        LedOn( LED2 );                          /* LED2 点灯     */
        SetRelAlarm( LedOffAlm, ALARM_CNT, 0 ); /* LED 消灯アラーム設定 */
        TerminateTask();                        /* 自タスク終了 */
}

/****************************************************************************/
/* 関数名 : TASK( LedOffTask )                                              */
/* 役割   : LED 消灯                                                         */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/****************************************************************************/
TASK( LedOffTask )
{
        LedOff( LED2 );                     /* LED2 消灯        */
        TerminateTask();                    /* 自タスク終了     */
}

/****************************************************************************/
/* 関数名 : void StartupHook( void )                                        */
/* 役割   : 各デバイスドライバの初期化                                       */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/****************************************************************************/
#ifdef USE_STARTUPHOOK
void StartupHook( void )
{
        InitT100usTimer();                  /* システムタイマ起動   */
        LedInit();                          /* LED 初期化           */
}       /* StartupHook     */
#endif /* USE_STARTUPHOOK */
```

11.1.4　プログラムの動作

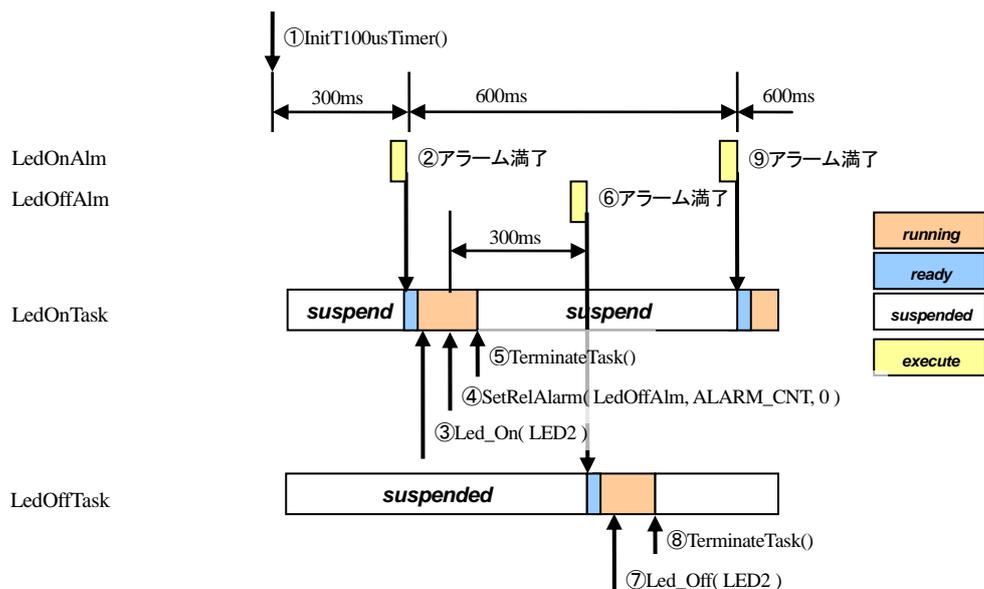



図 11.1-1 プログラムの動作

① 100μs タイマを起動します。
② ①から 300ms 経過すると LedOnAlam が満了し、LedOnTask を実行します。LedOnTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。
③ LedOnTask が実行状態に遷移した直後に LED3 を点灯します。
④ LedOffAlm を実行します。
⑤ LedOnTask は自分自身を終了し、休止状態に遷移します。
⑥ ④から 300ms 経過すると LedOffAlam が満了し、LedOffTask を実行します。LedOffTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。
⑦ LedOffTask が実行状態に遷移した直後に LED3 を消灯します。
⑧ LedOffTask は自分自身を終了し、休止状態に遷移します。
⑨ ②から 600ms 経過すると LedOnAlam が満了し、LedOnTask を実行します。LedOnTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。

以降、③～⑨の動作を繰り返します。

11.2 6.4.4 章 - リソース

11.2.1 OIL ファイル

【sample.oil】

```
#include "implementation.oil"

CPU current {
#include <t100us_timer.oil>

    OS os {
        STATUS = STANDARD;
        STARTUPHOOK = TRUE;
        ERRORHOOK = FALSE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        USEGETSERVICEID = TRUE;
        USEPARAMETERACCESS = TRUE;
        USERESSCHEDULER = FALSE;
    };

    APPMODE AppMode1 {};

    TASK LargeStrTask {
        AUTOSTART = FALSE;
        PRIORITY = 13;
        STACKSIZE = 0x0100;
        ACTIVATION = 1;
        SCHEDULE = FULL;
        RESOURCE = LcdRes;

    };

    TASK SmallStrTask {
        AUTOSTART = FALSE;
        PRIORITY = 14;
        STACKSIZE = 0x0100;
        ACTIVATION = 1;
        SCHEDULE = FULL;
        RESOURCE = LcdRes;
    };

    ALARM LargeStrAlm{
        COUNTER = T100usTimerCnt;
        ACTION  = ACTIVATETASK { TASK = "LargeStrTask"; };
        AUTOSTART = TRUE {
                    APPMODE = AppMode1;
                    ALARMTIME = 5000;
                    CYCLETIME = 5000;
                };
    };

    ALARM SmallStrAlm{
        COUNTER = T100usTimerCnt;
        ACTION  = ACTIVATETASK { TASK = "SmallStrTask"; };
        AUTOSTART = FALSE;
    };

    RESOURCE LcdRes {
```

```
        RESOURCEPROPERTY = STANDARD;
    };
};
```

11.2.2　ヘッダファイル

【sample.h】

```
/*****************************************************************************/
/*   インクルードファイル                                                          */
/*****************************************************************************/
#include "kernel_id.h"

/*****************************************************************************/
/*   定数定義                                                                  */
/*****************************************************************************/

/*****************************************************************************/
/*   外部宣言                                                                  */
/*****************************************************************************/
DeclareAlarm( SmallStrAlm );
DeclareResource( LcdRes );
DeclareTask( LargeStrTask );
DeclareTask( SmallStrTask );
```

11.2.3　ソースファイル

【sample.c】

```
/*****************************************************************************/
/*   インクルードファイル                                                          */
/*****************************************************************************/
#include "kernel.h"
#include "t100us_timer.h"
#include "sample.h"
#include "led.h"
#include "serial.h"
#include "lcd.h"

/*****************************************************************************/
/*   定数定義                                                                  */
/*****************************************************************************/
/* リソースを使用する場合は有効にする */
/* #define USE_LCD_RESOURCE */

#define ALARM_CNT ((UINT32)4ul)

/*****************************************************************************/
/*   変数定義                                                                  */
/*****************************************************************************/
static UINT8    lcd_str[ LCD_DEV_LINE ][ LCD_DEV_DIGIT+1 ]= {
    "ABCDEFGHIJKLMNOP",
    "abcdefghijklmnop",
};

/****************************************************************************/
/*   外部宣言                                                                 */
/****************************************************************************/
```

```
/*****************************************************************************/
/*   内部宣言                                                                */
/*****************************************************************************/
void main( void );
TASK( SmallStrTask );
TASK( LargeStrTask );

/*****************************************************************************/
/* 関数名 : void main( void )                                                */
/* 役割   : メイン処理                                                       */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/*****************************************************************************/
void main( void )
{
    StartOS( AppMode1 );                        /* OS スタート */
}

/*****************************************************************************/
/* 関数名 : TASK( LargeStrTask )                                             */
/* 役割   : LCD に大文字表示                                                 */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/*****************************************************************************/
TASK( LargeStrTask )
{
    LedOn( LED2 );                              /* LED2 点灯          */

    SetRelAlarm( SmallStrAlm, ALARM_CNT, 0 );   /* LED 消灯アラーム設定 */

#ifdef USE_LCD_RESOURCE
    GetResource( LcdRes );                      /* リソース獲得       */
#endif /* USE_LCD_RESOURCE */

    LcdWriteLine( 0, lcd_str[0], 1 );           /* LCD 書込み         */

#ifdef USE_LCD_RESOURCE
    ReleaseResource( LcdRes );                  /* リソース開放       */
#endif /* USE_LCD_RESOURCE */

    LedOff( LED2 );                             /* LED3 消灯          */

    TerminateTask();                            /* 自タスク終了       */
}

/*****************************************************************************/
/* 関数名 : TASK( SmallStrTask )                                             */
/* 役割   : LCD に小文字表示                                                 */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/*****************************************************************************/
TASK( SmallStrTask )
{
    LedOn( LED3 );                       /* LED3 点灯          */

#ifdef USE_LCD_RESOURCE
    GetResource( LcdRes );               /* リソース獲得       */
#endif /* USE_LCD_RESOURCE */

    LcdWriteLine( 0, lcd_str[1], 1 );    /* LCD 書込み         */

#ifdef USE_LCD_RESOURCE
    ReleaseResource( LcdRes );           /* リソース開放       */
```

```
#endif /* USE_LCD_RESOURCE */

    LedOff( LED3 );                     /* LED3 消灯          */

    TerminateTask();                    /* 自タスク終了       */
}

/*****************************************************************************/
/* 関数名 : void StartupHook( void )                                         */
/* 役割   : 各デバイスドライバの初期化                                       */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/*****************************************************************************/
#ifdef USE_STARTUPHOOK
void StartupHook( void )
{
    InitT100usTimer();                        /* システムタイマ起動    */
    LedInit();                                /* LED 初期化            */
    LcdInit();                                /* LCD 初期化            */
    LcdCtlDisplay( 0, LCD_CTL_CLRDISPLAY );   /* LCD クリア            */
}    /* StartupHook    */
#endif /* USE_STARTUPHOOK */
```

11.2.4 プログラムの動作

■リソースを使用しない場合

Sample.h の「#define USE_LCD_RESOURCE」をコメントアウトしている場合の動作を示します。

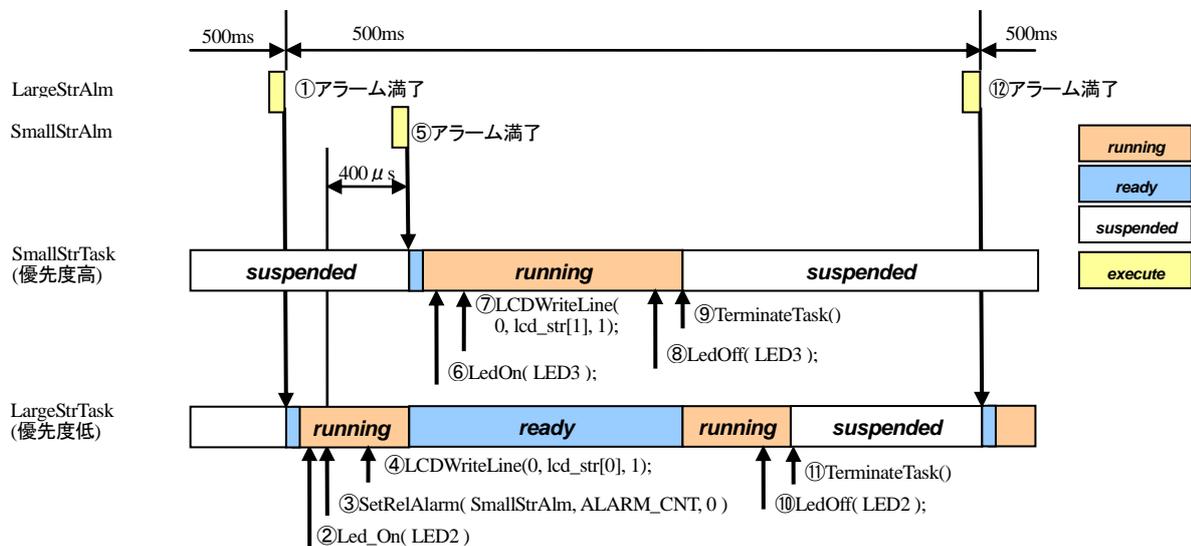



図 11.2-1 リソース未使用プログラムの動作

① LargeStrAlm が満了し、LargeStrTask を実行します。LargeStrTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。
② LargeStrTask が実行状態に遷移した直後に LED2 を点灯します。
③ SmallStrAlm を実行します。
④ LCD に文字列「ABCDEFGHIJKLMNOP」を書き出し始めます。
⑤ ③から 400μs 経過すると SmallStrAlm が満了し、SmallStrTask を実行します。SmallStrTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。LargeStrTask は実行状態から実行可能状態に遷移します。このとき LCD に文字列「ABCDEFGHIJKLMNOP」は書き出し終えていません。

⑥ SmallStrTask が実行状態に遷移した直後に LED3 を点灯します。
⑦ LCD に文字列「abcdefghijklmnop」を書き出し始めます。
⑧ LCD に文字列「abcdefghijklmnop」を書き出し終えた後、LED3 を消灯します。
⑨ SmallStrTask は自分自身を終了し、休止状態に遷移します。SmallStrTask が休止状態になると実行可能状態に遷移していた LargeStrTask が実行状態になり、LCD に書き出し終えていなかった文字列「ABCDEFGHIJKLMNOP」の残りを書き出します。このため、LCD には「ABCDEFGHIJKLMNOP」と「abcdefghijklmnop」が混じった文字列が表示されます。
⑩ LCD に文字列「ABCDEFGHIJKLMNOP」を書き出し終えた後、LED2 を消灯します。
⑪ LargeStrTask は自分自身を終了し、休止状態に遷移します。
⑫ ①から 500ms 経過すると LargeStrAlm が満了し、LargeStrTask を実行します。LargeStrTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。

以降、②～⑫の動作を繰り返します。

■リソースを使用する場合

Sample.h の「#define USE_LCD_RESOURCE」をコメントアウトしない場合の動作を示します。

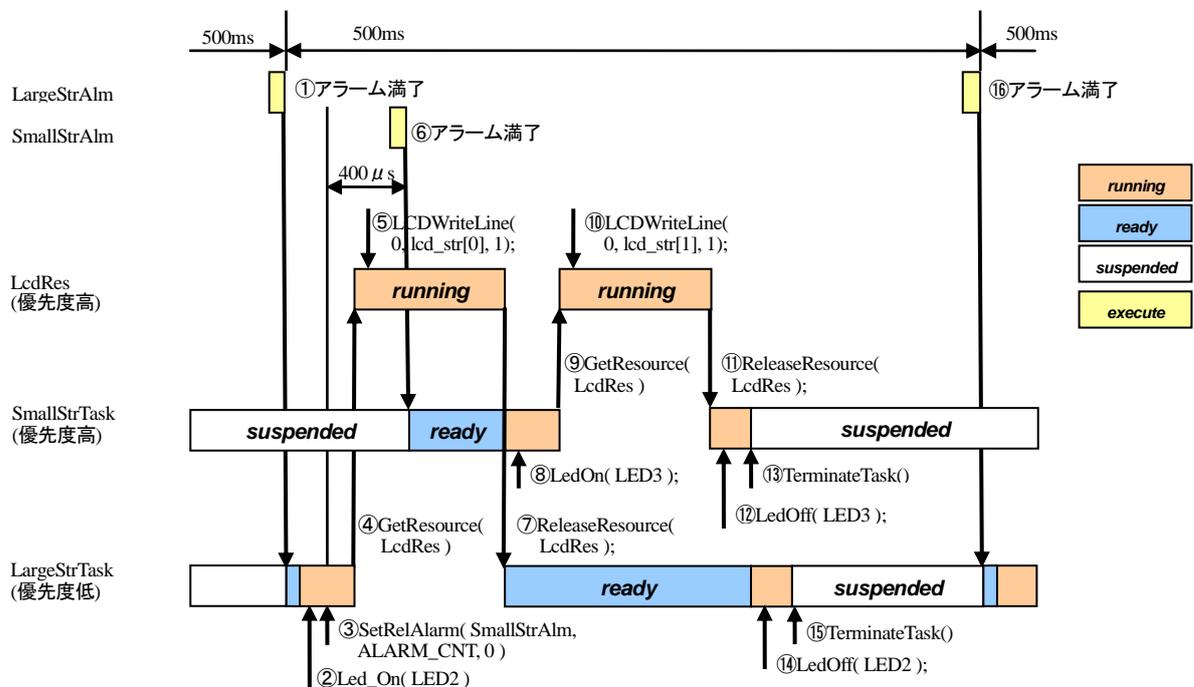



図 11.2-2 リソース使用プログラムの動作

① LargeStrAlm が満了し、LargeStrTask を実行します。LargeStrTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。
② LargeStrTask が実行状態に遷移した直後に LED2 を点灯します。
③ SmallStrAlm を実行します。
④ LargeStrTask が LcdRes を獲得すると LargeStrTask の優先度は SmallStrTask と同一優先度になります。
⑤ LCD に文字列「ABCDEFGHIJKLMNOP」を書き出し始めます。
⑥ ③から 400μs 経過すると SmallStrAlm が満了し、SmallStrTask を実行します。SmallStrTask は休止状態から実行可能状態に遷移します。LcdRes を獲得している LargeStrTask は SmallStrTask と同一優先度のため実行状態を継続します。
⑦ LCD に文字列「ABCDEFGHIJKLMNOP」を書き出し終えた後、LcdRes を解放します。SmallStrTask は LargeStrTask より優先度が高くなるため実行状態になり、LargeStrTask は実行可能状態になります。
⑧ SmallStrTask が実行状態に遷移した直後に LED3 を点灯します。
⑨ SmallStrTask が LcdRes を獲得します。LcdRes を使用するタスクの中で SmallStrTask が一番高優先度なた

め、SmallStrTask の優先度は変わりません。

⑩ LCD に文字列「abcdefghijklmnop」を書き出し始めます。

⑪ LCD に文字列「abcdefghijklmnop」を書き出し終えた後、LcdRes を解放します。

⑫ SmallStrTask の終了直前に LED3 を消灯します。

⑬ SmallStrTask は自分自身を終了し、休止状態に遷移します。SmallStrTask が休止状態になると実行可能状態に遷移していた LargeStrTask が実行状態になります。リソースを使用しない場合と異なり、文字列「ABCDEFGHIJKLMNOP」の書き出しを終えた後に文字列「abcdefghijklmnop」を書き出したので、LCD には文字列「abcdefghijklmnop」が表示されます。

⑭ LargeStrTask の終了直前に LED2 を消灯します。

⑮ LargeStrTask は自分自身を終了し、休止状態に遷移します。

⑯ ①から 500ms 経過すると LargeStrAlm が満了し、LargeStrTask を実行します。LargeStrTask は休止状態から実行可能状態に遷移し、実行状態に遷移します。

以降、②～⑯の動作を繰り返します。

11.3 7.3 章 - 演習問題１

```
unsigned int g_status;                          /* ルール 5.2 識別子の隠ぺい（必要） */

#define STATE1      ((unsigned int)(0U))
#define STATE2      ((unsigned int)(1U))

void func(void)
{
        unsigned int status = 0;
        unsigned int input = 0;                 /* ルール 9.1 自動変数（必要）*/


        /* func2 は外部に宣言されている関数とする */
        if( func2( input ) == 0x02 ) {          /* ルール 7.1 （0 以外の）8 進定数（必要） */
                status = STATE1;
        }
        else {                                  /* ルール 14.9 “if（式）”の後（必要） */
                status = STATE2;
        }
        return;
}
```

11.4 7.3 章 - 演習問題２

```
#define ARRAY_SIZE          (100)

void func(void)
{
        int i;                                  /* ルール6.1 char型（必要） */
        unsigned int array[ARRAY_SIZE];

        for( i = 0 ; i < ARRAY_SIZE ; i++ ) {   /* ルール14.8 繰り返し文の本体（必要） */
                array[i] = i;
        }
        return;
}
```

11.5 8.2.8 章 - シリアル通信プログラムの作成

11.5.1 OIL ファイル

【sample.oil】

```
#include "implementation.oil"
CPU current {

        #include "serial.oil"
        #include "t100us_timer.oil"

        OS os {
                STARTUPHOOK = FALSE;
                ERRORHOOK = FALSE;
                SHUTDOWNHOOK = FALSE;
                PRETASKHOOK = FALSE;
                POSTTASKHOOK = FALSE;
                USERESSCHEDULER = TRUE;
        };

        APPMODE AppMode1 {};

        TASK MainTask {
                AUTOSTART = TRUE {
                        APPMODE = AppMode1;
                };
                PRIORITY = 8;
                ACTIVATION = 1;
                SCHEDULE = NON;
        };

        TASK SendTask {
                PRIORITY = 7;
                ACTIVATION = 1;
                SCHEDULE = FULL;
                EVENT = RxFinishEvt;
        };

        TASK ReceiveTask {
                AUTOSTART = FALSE;
                PRIORITY = 6;
                ACTIVATION = 1;
                SCHEDULE = FULL;
        };

        EVENT RxFinishEvt {
                MASK = AUTO;
        };
};
```

11.5.2 ヘッダファイル

【sample.h】

```
/*****************************************************************************/
/*   インクルードファイル                                                    */
/*****************************************************************************/
#include "kernel_id.h"
```

```
/*****************************************************************************/
/*   定数定義                                                                 */
/*****************************************************************************/
#define SERIAL_BUF_MAX       ( 64 )    /* シリアル受信バッファ最大値 */

/*****************************************************************************/
/*   外部宣言                                                                 */
/*****************************************************************************/
DeclareTask( MainTask );
```

11.5.3 ソースファイル

【sample.c】

```
/*****************************************************************************/
/*   インクルードファイル                                                     */
/*****************************************************************************/
#include "kernel.h"
#include "kernel_id.h"
#include "serial.h"
#include "sample.h"

/*****************************************************************************/
/*   定数定義                                                                 */
/*****************************************************************************/

/*****************************************************************************/
/*   変数定義                                                                 */
/*****************************************************************************/
static char serial_buf[ SERIAL_BUF_MAX ] = "";
static int serial_len = 0;

/****************************************************************************/
/*   外部宣言                                                                */
/****************************************************************************/

/*****************************************************************************/
/*   内部宣言                                                                 */
/*****************************************************************************/
void main( void );
DeclareTask( ReceiveTask );
DeclareTask( SendTask );
DeclareEvent( RxFinishEvt );
TASK( MainTask );
TASK( ReceiveTask );
TASK( SendTask );

/****************************************************************************/
/* 関数名 : int main( void )                                                 */
/* 役割   : カーネル起動                                                     */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/* 備考   : 無し                                                             */
/****************************************************************************/
void main( void )
{
    /* カーネル起動 */
    StartOS( AppMode1 );
}

/****************************************************************************/
```

```
/* 関数名 : TASK( MainTask )                                                  */
/* 役割   : 初期化処理実施後、                                                */
/*          シリアル受信タスク、シリアル送信タスクを起動する。                */
/* 戻り値 : 無し                                                              */
/* 引数   : 無し                                                              */
/* 備考   : 無し                                                              */
/*****************************************************************************/
TASK( MainTask )
{
    /* シリアル初期化 */
    InitSerial();
    /* 受信タスク起動 */
    ActivateTask( ReceiveTask );
    /* 送信タスク起動 */
    ActivateTask( SendTask );

    /* タスク終了 */
    TerminateTask( );
}

/*****************************************************************************/
/* 関数名 : TASK( SendTask )                                                  */
/* 役割   : バッファに格納されたメッセージをシリアル出力する。                */
/* 戻り値 : 無し                                                              */
/* 引数   : 無し                                                              */
/* 備考   : RxFinishEvt イベント発生時に起動する。                            */
/*****************************************************************************/
TASK( SendTask )
{
    while( 1 ) {
        /* 受信完了イベント待ち */
        WaitEvent( RxFinishEvt );
        /* イベントクリア */
        ClearEvent( RxFinishEvt );
        /* 格納文字列出力 */
        SendSerialStr( serial_buf );
        /* 改行コード出力 */
        SendSerialStr( "¥n" );

        /* 受信数クリア */
        serial_len = 0;
    }
    /* タスク終了 */
    TerminateTask( );
}

/*****************************************************************************/
/* 関数名 : TASK(ReceiveTask)                                                 */
/* 役割   : キー入力された英数字の文字数がシリアル受信バッファ最大値を        */
/*          超えるか、Enter 入力を受けた時点で入力値をバッファに格納する      */
/* 戻り値 : 無し                                                              */
/* 引数   : 無し                                                              */
/* 備考   : バッファ格納が完了すると RxFinishEvt イベントを発生させる。       */
/*****************************************************************************/
TASK(ReceiveTask)
{
    /* キー入力文字格納用変数 */
    char rcv_char = '¥0';

    while( 1 ) {
        /* バッファが限界値になるか、改行を受信するまでループ */
        do {
            /* 文字取得 */
            RecvPolSerialChar( &rcv_char );
```

```
            /* 終端文字以外のとき */
            if( rcv_char != '¥0' ) {
                /* バッファに代入 */
                serial_buf[ serial_len ] = rcv_char;
                serial_len++;
            }
        } while ( ( serial_len < ( SERIAL_BUF_MAX - 2 )) &&
                  ( rcv_char != '¥r' ) );

        if( serial_len == ( SERIAL_BUF_MAX - 2 )) {
            serial_buf[ serial_len ]     = '¥r';
            serial_buf[ serial_len + 1 ] = '¥0';
        }
        else {
            /* 終端文字を代入 */
            serial_buf[ serial_len ] = '¥0';
        }
        /* 受信完了イベントを発生させる */
        SetEvent( SendTask , RxFinishEvt );
    }

    /* 自分を終了して受信タスク起動 */
    TerminateTask( );
}
```

11.6 8.4 章 - 提供デバイスドライバを用いたアプリケーションの作成

11.6.1 OIL ファイル

【sample.oil】

```
/*
 *  定義部のインクルード
 */
#include "implementation.oil"


/*
 *  実装部
 */
CPU current {

/* syslib のシリアル機能を使用する */
#include <serial.oil>

/* システムタイマ機能を使用する */
#include <t100us_timer.oil>

    /*
     *  OS 定義
     */
    OS os {
        STATUS = STANDARD;
        STARTUPHOOK = TRUE;
        ERRORHOOK = FALSE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        USEGETSERVICEID = TRUE;
        USEPARAMETERACCESS = TRUE;
        USERESSCHEDULER = TRUE;
    };

    /*
     *  アプリケーションモード定義
     */
    APPMODE AppMode1 {};

    /*
     *  タスク定義
     */
    /* スイッチタスク    */
    TASK SwitchTask {
        AUTOSTART = FALSE;
        PRIORITY = 6;
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };

    /* 表示タスク */
    TASK DispTask {
        AUTOSTART = TRUE
        {
            APPMODE = AppMode1;
        };
        PRIORITY = 7;
```

```
        ACTIVATION = 1;
        SCHEDULE = FULL;
        EVENT = SwPushEvt;
    };



    EVENT SwPushEvt {
        MASK = AUTO;
    };

    /* スイッチタスク周期アラーム */
    ALARM SwitchCycArm {
        COUNTER = T100usTimerCnt;
        ACTION  = ACTIVATETASK {
            TASK = SwitchTask;
        };
        AUTOSTART = TRUE {
            APPMODE = AppMode1;
            ALARMTIME = 9999;
            CYCLETIME = 200;
        };
    };
};
```

11.6.2　ヘッダファイル

【sample.h】

```
/******************************************************************************/
/*   インクルードファイル                                                      */
/******************************************************************************/
#include "kernel_id.h"

/******************************************************************************/
/*   定数定義                                                                  */
/******************************************************************************/
/* LED 状態変数操作用マクロ */
#define LED_STATE_CLEAR    ( UINT8 )( 0x00 )

/* LED 点灯・消灯 */
#define LED_OFF    ( UINT8 )( 0 )
#define LED_ON    ( UINT8 )( i )

/* LCD への LED 状態表示パターン */
#define LED_STATE_PATERN    ( UINT8 )16

/******************************************************************************/
/*   外部宣言                                                                  */
/******************************************************************************/
DeclareTask( SwitchTask );
DeclareTask( DispTask );
DeclareEvent( SwPushEvt );
```

11.6.3　ソースファイル

【sample.c】

```
/******************************************************************************/
/*   インクルードファイル                                                     */
/******************************************************************************/
#include "kernel.h"
#include "t100us_timer.h"
#include "kernel_id.h"
#include "sample.h"
#include "sw.h"
#include "led.h"
#include "lcd.h"

/******************************************************************************/
/*   定数定義                                                                 */
/******************************************************************************/

/******************************************************************************/
/*   変数定義                                                                 */
/******************************************************************************/
/* LED 状態保持変数 */
static UINT8 led_state_buff;
/* LCD 段目見出しメッセージ文字列 */
static const UINT8 lcd_headline[]    = " 5 / 4 / 3 / 2  ";
static const UINT8 lcd_state_led[ LED_STATE_PATERN ][] = { "OFF/OFF/OFF/OFF",
                                                           "OFF/OFF/OFF/ON ",
                                                           "OFF/OFF/ON /OFF",
                                                           "OFF/OFF/ON /ON ",
                                                           "OFF/ON /OFF/OFF",
                                                           "OFF/ON /OFF/ON ",
                                                           "OFF/ON /ON /OFF",
                                                           "OFF/ON /ON /ON ",
                                                           "ON /OFF/OFF/OFF",
                                                           "ON /OFF/OFF/ON ",
                                                           "ON /OFF/ON /OFF",
                                                           "ON /OFF/ON /ON ",
                                                           "ON /ON /OFF/OFF",
                                                           "ON /ON /OFF/ON ",
                                                           "ON /ON /ON /OFF",
                                                           "ON /ON /ON /ON ",
                                                         };

/******************************************************************************/
/*   外部宣言                                                                 */
/******************************************************************************/

/******************************************************************************/
/*   内部宣言                                                                 */
/******************************************************************************/
void main( void );
TASK( SwitchTask );
TASK( DispTask );

/******************************************************************************/
/* 関数名 : void main( void )                                                 */
/* 役割   : メイン処理                                                        */
/* 戻り値 : 無し                                                              */
/* 引数   : 無し                                                              */
/* 備考   : 無し                                                              */
/******************************************************************************/
void main( void )
{
    /*
     *  カーネル起動
     */
    StartOS( AppMode1 );
```

```
}

/****************************************************************************/
/* 関数名 : TASK( SwitchTask )                                              */
/* 役割   : スイッチ押下状態監視タスク                                      */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/* 備考   : 20ms 周期で起動し、SW3～SW6 の押下状態を取得する。              */
/*          SW の押下状態が変化していたとき sw_push イベントを発生させる。  */
/****************************************************************************/
TASK( SwitchTask )
{
    /* スイッチ状態取得用変数 */
    UINT8 sw_state[4];
    static UINT8 pre_sw_state[4] =
            { SW_OFF , SW_OFF , SW_OFF , SW_OFF };

    /* 各スイッチの状態を取得 */
    SwGetPush3( &sw_state[0] );
    SwGetPush4( &sw_state[1] );
    SwGetPush5( &sw_state[2] );
    SwGetPush6( &sw_state[3] );

    /* スイッチが押されている場合、LED 状態変数に or 代入していく */
    if( ( SW_ON == sw_state[0] && pre_sw_state[0] == SW_OFF ) ||
        ( SW_OFF == sw_state[0] && pre_sw_state[0] == SW_ON ) ) {
        led_state_buff |= LED5;
    }
    if( ( SW_ON == sw_state[1] && pre_sw_state[1] == SW_OFF ) ||
        ( SW_OFF == sw_state[1] && pre_sw_state[1] == SW_ON ) ) {
        led_state_buff |= LED4;
    }
    if( ( SW_ON == sw_state[2] && pre_sw_state[2] == SW_OFF ) ||
        ( SW_OFF == sw_state[2] && pre_sw_state[2] == SW_ON ) )    {
        led_state_buff |= LED3;
    }
    if( ( SW_ON == sw_state[3] && pre_sw_state[3] == SW_OFF ) ||
        ( SW_OFF == sw_state[3] && pre_sw_state[3] == SW_ON ) )    {
        led_state_buff |= LED2;
    }

    /* LED がクリア状態で無い場合、イベントを発生 */
    if( led_state_buff != LED_STATE_CLEAR ) {
        SetEvent( DispTask, SwPushEvt );
    }

    /* 状態の保存 */
    pre_sw_state[0] = sw_state[0];
    pre_sw_state[1] = sw_state[1];
    pre_sw_state[2] = sw_state[2];
    pre_sw_state[3] = sw_state[3];

    TerminateTask();
}

/****************************************************************************/
/* 関数名 : TASK( DispTask )                                                */
/* 役割   : LED、LCD 表示タスク                                             */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/* 備考   : sw_push イベント発生時に起動する。                              */
/*          sw 押下状態を見て LED、LCD に状態を表示する。                   */
/****************************************************************************/
```

```
TASK( DispTask )
{
    UINT8 led_patern = 0x00;

    while( 1 ) {
        WaitEvent( SwPushEvt );
        ClearEvent( SwPushEvt );

        /* LED を点灯・消灯する */
        LedRev( led_state_buff );
        led_patern ^= led_state_buff;

        /* LED の状態を LCD で表示 */
        LcdWriteLine( 0, lcd_state_led[ ( led_patern >> 4 ) ], 2 );

        /* LED の状態をクリア */
        led_state_buff = LED_STATE_CLEAR;
    }

    TerminateTask( );
}

/****************************************************************************/
/* 関数名 : void StartupHook( void )                                        */
/* 役割   : 各デバイスドライバ初期化                                        */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/* 備考   : 無し                                                            */
/****************************************************************************/
#ifdef USE_STARTUPHOOK
void StartupHook( void )
{
    /* システムタイマ初期化 */
    InitT100usTimer();

    /* LED 初期化 */
    LedInit();

    /* スイッチ初期化 */
    SwInit();

    /* LCD 初期化 */
    LcdInit();

    /* LCD クリア */
    LcdCtlDisplay( 0, LCD_CTL_CLRDISPLAY );

    /* LCD 一行目表示 */
    LcdWriteLine( 0, lcd_headline, 1 );

    /* LCD 二行目初期表示 */
    LcdWriteLine( 0, lcd_state_led[0], 2 );

}    /* StartupHook    */
#endif /* USE_STARTUPHOOK */
```

11.7 9.1 章 – CAN 通信プロトコル

ＩＤ ＝0x18DB33F1

11.8 9.2 章 – CAN 通信ドライバの開発

11.8.1 ヘッダファイル

【can.h】

```
#ifndef CAN_H_
#define CAN_H_

/******************************************************************************/
/*   インクルードファイル                                                        */
/******************************************************************************/
#include "kernel.h"
#include "kernel_id.h"

/******************************************************************************/
/*   定数定義                                                                  */
/******************************************************************************/
/* エラーコード */
#define CAN_E_OK            ((UINT8)0x00)
#define CAN_E_PRM           ((UINT8)0x01)           /* 引数異常 */
#define CAN_E_RUNNING       ((UINT8)0x02)           /* 送信中or受信中 */
#define CAN_E_OVERRUN       ((UINT8)0x03)           /* オーバランエラー */

#define CAN_SID_MAX         ((UINT16)0x07ff)    /* 標準IDの最大値 */
#define CAN_DLC_MAX         ((UINT8)0x08)           /* 送信データの最大バイト数 */

/******************************************************************************/
/*   変数定義                                                                  */
/******************************************************************************/

/******************************************************************************/
/*   外部宣言                                                                  */
/******************************************************************************/

/******************************************************************************/
/* 関数名 : void CanInit( void )                                                */
/* 役割   : CAN0スロットの初期化                                                 */
/* 戻り値 : 無し                                                                */
/* 引数   : 無し                                                                */
/* 備考   : 本デバイスドライバを使用する際には一番最初に呼び出すこと                */
/******************************************************************************/
extern void CanInit( void );

/******************************************************************************/
/* 関数名 : UINT8 CanSetTrm( UINT16 id, UINT8 dlc, UINT8 *data )              */
/* 役割   : 引数異常を確認後、CAN送信設定依存部関数を呼び出して、                 */
/*          送信設定処理を行う                                                  */
/* 戻り値 : UINT8 ret                                                           */
/*            CAN_E_OK      - 正常終了                                          */
/*            CAN_E_PRM     - 引数が不正                                        */
/*            CAN_E_RUNNNIG - 送信処理中                                        */
/* 引数   : UINT16 id       - 送信する標準ID ( 0x0000 - 0x07FF)                 */
/*          UINT8  dlc      - 送信データ長 ( 0 - 8 )                            */
/*          UINT8  *data    - 送信データを格納した配列の先頭ポインタ              */
```

```
/* 備考   : 初期化関数処理終了後に呼び出すこと                              */
/*****************************************************************************/
extern UINT8 CanSetTrm( UINT16 id, UINT8 dlc, UINT8 *data );

/*****************************************************************************/
/* 関数名 : UINT8 CanSetRec( UINT16 id, UINT8 slot )                          */
/* 役割   : 引数異常を確認後、CAN受信設定依存部関数を呼び出して、             */
/*          受信設定処理を行う                                                */
/* 戻り値 : UINT8 ret                                                         */
/*            CAN_E_OK      - 正常終了                                        */
/*            CAN_E_PRM     - 引数が不正                                      */
/*            CAN_E_RUNNNIG - 送信処理中                                      */
/* 引数   : UINT16 id       - 受信設定する標準ID ( 0x0000 - 0x07FF )          */
/*          UINT8  slot     - メッセージスロット ( 1 - 15 )                   */
/* 備考   : 初期化関数処理終了後に呼び出すこと                              */
/*****************************************************************************/
extern UINT8 CanSetRec( UINT16 id , UINT8 slot );

/*****************************************************************************/
/* 関数名 : void CanRecCbr( UINT8 rec_err, UINT16 id UINT8 dlc, UINT8 *data )*/
/* 役割   : CAN受信割り込みから呼び出される                                  */
/*          CAN受信完了を通知するコールバック関数                            */
/*          引数で正常受信orオーバランエラー発生、受信完了したID、          */
/*          受信データ長、受信データを格納した配列の先頭アドレスを渡す      */
/*          本関数内での処理内容は上位層に一任する                          */
/* 戻り値 : 無し                                                              */
/* 引数   : UINT8  rec_err - 受信成否通知( CAN_E_OK, CAN_E_OVERRUN_E )        */
/*          UINT16 id      - 受信ID( 0x0000 - 0x07FF )                        */
/*          UINT8  dlc     - 受信データ長 ( 0 - 8 )                           */
/*          UINT8  *data   - 受信データ格納した配列の先頭アドレス            */
/* 備考   : CAN受信割り込み依存部関数hw_can_rec_intによって呼び出される      */
/*****************************************************************************/
extern void CanRecCbr( UINT8 rec_err, UINT16 id, UINT8 dlc, UINT8 *data );

#endif /* CAN_H_ */
```

11.8.2　ソースファイル

【can.h】

```
/*****************************************************************************/
/*   インクルードファイル                                                    */
/*****************************************************************************/
#include "hw_can.h"

/*****************************************************************************/
/*   定数定義                                                                */
/*****************************************************************************/

/*****************************************************************************/
/*   変数定義                                                                */
/*****************************************************************************/

/*****************************************************************************/
/*   外部宣言                                                                */
/*****************************************************************************/



/*****************************************************************************/
/* 関数名 : void CanInit( void )                                              */
/* 役割   : CAN0スロットの初期化                                              */
/* 戻り値 : 無し                                                              */
```

```
/* 引数   : 無し                                                        */
/* 備考   : 本デバイスドライバを使用する際には一番最初に呼び出すこと        */
/*****************************************************************************/
void CanInit( void )
{
        /* CAN初期化依存部関数を呼び出し */
        hw_can_init();
}

/*****************************************************************************/
/* 関数名 : UINT8 CanSetTrm( UINT16 id, UINT8 dlc, UINT8 *data )           */
/* 役割   : 引数異常を確認後、CAN送信設定依存部関数を呼び出して、          */
/*          送信設定処理を行う                                              */
/* 戻り値 : UINT8 ret                                                       */
/*            CAN_E_OK      - 正常終了                                      */
/*            CAN_E_PRM     - 引数が不正                                    */
/*            CAN_E_RUNNNIG - 送信処理中                                    */
/* 引数   : UINT16 id       - 送信する標準ID ( 0x0000 - 0x07FF)             */
/*          UINT8  dlc      - 送信データ長 ( 0 - 8 )                        */
/*          UINT8  *data    - 送信データを格納した配列の先頭ポインタ        */
/* 備考   : 初期化関数処理終了後に呼び出すこと                              */
/*****************************************************************************/
UINT8 CanSetTrm( UINT16 id, UINT8 dlc, UINT8 *data )
{
        UINT8     ret = CAN_E_OK;     /* 戻り値初期化 */
        UINT8     cnt;

        /* 引数の確認 */
        if(( id > CAN_SID_MAX ) ||
           ( dlc > CAN_DLC_MAX )) {
                ret = CAN_E_PRM;
        }
        else {
                ret = hw_can_set_trm( id, dlc, data );
        }

        return ret;
}

/*****************************************************************************/
/* 関数名 : UINT8 CanSetRec( UINT16 id, UINT8 slot )                        */
/* 役割   : 引数異常を確認後、CAN受信設定依存部関数を呼び出して、          */
/*          受信設定処理を行う                                              */
/* 戻り値 : UINT8 ret                                                       */
/*            CAN_E_OK      - 正常終了                                      */
/*            CAN_E_PRM     - 引数が不正                                    */
/*            CAN_E_RUNNNIG - 送信処理中                                    */
/* 引数   : UINT16 id       - 受信設定する標準ID ( 0x0000 - 0x07FF )        */
/*          UINT8  slot     - メッセージスロット ( 1 - 15 )                 */
/* 備考   : 初期化関数処理終了後に呼び出すこと                              */
/*****************************************************************************/
UINT8 CanSetRec( UINT16 id , UINT8 slot)
{
        UINT8     ret = CAN_E_OK;     /* 戻り値初期化 */

        /* 引数の確認 */
        if(( id > CAN_SID_MAX ) ||
           ( slot == 0 ) ||
           ( slot > 15 )) {
                ret = CAN_E_PRM;
        }
        else{
                ret = hw_can_set_rec( id , slot );
        }
```

```
        return ret;
}
```

11.8.3 依存部ヘッダファイル

【hw_can.h】

```
#ifndef HW_CAN_H_
#define HW_CAN_H_

/*****************************************************************************/
/*   インクルードファイル                                                     */
/*****************************************************************************/
#include "can.h"
#include "sfrm32c85.h"


/*****************************************************************************/
/*   定数定義                                                                 */
/*****************************************************************************/
#define CAN_HIGH            ((UINT8)1)
#define CAN_LOW             ((UINT8)0)
#define CAN_CLEAR_8B        ((UINT8)0x00)     /* 8bitクリア用  */
#define CAN_CLEAR_16B       ((UINT16)0x0000)  /* 16bitクリア用 */

#define CAN_RESET_MODE      ((UINT16)0x0011)  /* CANリセットモード */
#define CAN_RUN_MODE        ((UINT8)0x00)     /* CAN通常動作モード */
#define CAN_TRM_REQ         ((UINT8)0x80)     /* CAN送信要求 */
#define CAN_REC_REQ         ((UINT8)0x40)     /* CAN送信要求 */

#define CAN_SET_NEWDATA     ((UINT8)0x01)     /* 受信完了フラグビット設定 */
#define CAN_CLEAR_NEWDATA   ((UINT8)0xfe)     /* 受信完了フラグビットクリア */
#define CAN_SET_TRAMACTIVE  ((UINT8)0x02)     /* 送信中フラグビット設定 */
#define CAN_SET_INVALDATA   ((UINT8)0x02)     /* 受信中フラグビット設定 */

/* CAN制御レジスタ用マクロ */
#define CAN_LOOPBACK    ((UINT8)0)    /* ループバック   0:OFF / 1:ON */
#define CAN_BASICCAN    ((UINT8)0)    /* BasicCANモード 0:OFF / 1:ON */
/* タイムスタンププリスケーラ                         */
/* TSPRE0  | TSPRE1 |          機能                */
/*--------+--------+-------------------------------*/
/*   0    |   0    | CANバスビットクロック          */
/*   0    |   1    | CANバスビットクロックの2分周 */
/*   1    |   0    | CANバスビットクロックの3分周 */
/*   1    |   1    | CANバスビットクロックの4分周 */
#define CAN_TSPRE0     ((UINT8)0)
#define CAN_TSPRE1     ((UINT8)0)
#define CAN_INTSEL     ((UINT8)1)   /* CAN割り込みモード */
                                    /* 0:3種類のCAN割り込みを纏めて出力 */
                                    /* 1:3種類のCAN割り込みを分けて出力 */

/* ボーレート設定マクロ (計算方法はハードウェアマニュアル参照      */
#define CAN_BRP     ((UINT8)3)        /* ボーレートプリスケーラ    */
#define CAN_SAM     ((UINT8)1)        /* サンプリング数 0:1回 / 1:3回 */
/* PTS幅                                                     */
/* PTS2,PTS1,PTS0 = 000:1Tq / 001:2Tq / 010:3Tq / 011:4Tq */
/*                  100:5Tq / 101:6Tq / 110:7Tq / 111:8Tq */
#define CAN_PTS0    ((UINT8)0)
#define CAN_PTS1    ((UINT8)0)
#define CAN_PTS2    ((UINT8)1)
/* PBS1幅                                                        */
```

```
/* PBS12,PBS11,PBS10 = 000:NA  / 001:2Tq / 010:3Tq / 011:4Tq */
/*                     100:5Tq / 101:6Tq / 110:7Tq / 111:8Tq */
#define CAN_PBS10    ((UINT8)1)
#define CAN_PBS11    ((UINT8)0)
#define CAN_PBS12    ((UINT8)1)
/* PBS2幅                                                    */
/* PBS22,PBS21,PBS20 = 000:NA  / 001:2Tq / 010:3Tq / 011:4Tq */
/*                     100:5Tq / 101:6Tq / 110:7Tq / 111:8Tq */
#define CAN_PBS20    ((UINT8)1)
#define CAN_PBS21    ((UINT8)1)
#define CAN_PBS22    ((UINT8)0)
/* SW幅                                        */
/* SJW1,SJW0 = 00:1Tq / 01:2Tq / 10:3Tq / 11:4Tq */
#define CAN_SJW0    ((UINT8)0)
#define CAN_SJW1    ((UINT8)1)

/* アクセプタンスフィルタマスク設定用マクロ                         */
/* 標準ID用マクロは標準ID0～ID10、拡張ID用マクロは拡張ID0～ID17が、 */
/* 各マクロの値の下位ビットから対応し、値を1としたIDに対して         */
/* アクセプタンスフィルタが行われる。                               */
/* グローバルマスク(メッセージバッファ1～13用標準ID)設定マクロ      */
#define CAN_GM_SID        ((UINT16)0x07ff)
/* グローバルマスク(メッセージバッファ1～13用拡張ID)設定マクロ      */
#define CAN_GM_EID        ((UINT32)0x0003ffff)
/* ローカルマスクA (メッセージバッファ14用   標準ID)設定マクロ      */
#define CAN_LMA_SID        ((UINT16)0x07ff)
/* ローカルマスクA (メッセージバッファ14用   拡張ID)設定マクロ      */
#define CAN_LMA_EID        ((UINT32)0x0003ffff)
/* ローカルマスクB (メッセージバッファ15用   標準ID)設定マクロ      */
#define CAN_LMB_SID        ((UINT16)0x07ff)
/* ローカルマスクB (メッセージバッファ15用   拡張ID)設定マクロ      */
#define CAN_LMB_EID        ((UINT32)0x0003ffff)

/* 拡張ID設定マクロ                                                  */
/* 各ビットが下位からメッセージスロット0～15に対応。0:標準ID / 1:拡張ID */
#define CAN_SET_ID        ((UINT16)0x0000)

/* スロット割り込みマスク設定用マクロ設定          */
/* 各ビットが下位からメッセージスロット0～15に対応 */
/* 0:割り込み禁止 / 1:割り込み許可                 */
#define CAN_SLOT_INT    ((UINT16)0xffff)

/* エラー割り込み設定用マクロ                             */
/* バスオフ状態遷移時の割り込み要求       0:不許可 / 1:許可 */
#define CAN_BUSOFF_INT    ((UINT8)0)
/* エラーパッシブ状態遷移時の割り込み要求 0:不許可 / 1:許可 */
#define CAN_ERRPAS_INT    ((UINT8)0)
/* バスエラーが発生時の割り込み要求       0:不許可 / 1:許可 */
#define CAN_BUSERR_INT    ((UINT8)0)

/* 拡張ID設定マクロ                                */
/* 各ビットが下位からメッセージスロット0～15に対応 */
/* 0:使用しない / 1:使用する                       */
#define CAN_SINGLE_S    ((UINT8)0x00)


/******************************************************************************/
/*   変数定義                                                                 */
/******************************************************************************/

/******************************************************************************/
/*   外部宣言                                                                 */
/******************************************************************************/
```

```
/***************************************************************************/
/* 関数名 : void hw_can_init( void )                                        */
/* 役割   : CAN0スロットの初期化                                             */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/* 備考   : 無し                                                             */
/***************************************************************************/
void hw_can_init( void );

/***************************************************************************/
/* 関数名 : UINT8 hw_can_set_trm( UINT16 id, UINT8 dlc, UINT8 *data )       */
/* 役割   : メッセージスロット0を通して、メッセージスロット0に               */
/*          送信ID・送信データ長・送信データを設定し、送信要求を出す。       */
/*          送信データは送信データ長で指定されたデータ数だけ設定する。       */
/* 戻り値 : UINT8 ret                                                        */
/*            CAN_E_OK      - 正常終了                                       */
/*            CAN_E_RUNNING - 送信中                                         */
/* 引数   : UINT16 id       - 送信する標準ID ( 0x0000 - 0x07FF)              */
/*          UINT8  dlc      - 送信データ長 ( 0 - 8 )                         */
/*          UINT8  *data    - 送信データを格納した配列の先頭ポインタ         */
/* 備考   : 無し                                                             */
/***************************************************************************/
UINT8 hw_can_set_trm( UINT16 id, UINT8 dlc, UINT8 *data );

/***************************************************************************/
/* 関数名 : UINT8 hw_can_set_rec( UINT16 id, UINT8 slot )                   */
/* 役割   : 指定されたメッセージスロットをスロットバッファ1を割り当てる     */
/*          スロットバッファ1を通して、メッセージスロットに受信IDを設定し、 */
/*          受信要求を出す                                                   */
/* 戻り値 : CAN_E_OK      : 正常終了                                         */
/*        : CAN_E_RUNNING : 受信中                                           */
/* 引数   : UINT16 id     : 受信設定する標準ID ( 0x0000 - 0x07FF )           */
/*          UINT8  slot   : 受信設定するメッセージスロット ( 1 - 15 )        */
/* 備考   : 無し                                                             */
/***************************************************************************/
UINT8 hw_can_set_rec( UINT16 id , UINT8 slot );

/***************************************************************************/
/* 関数名 : void hw_can_rec_int( void )                                     */
/* 役割   : メッセージスロット1の受信終了割り込みで起こされる               */
/*          データ長とデータを取得し、コールバックルーチンで上位層に渡す     */
/*          データ取得中にオーバランエラーが起きた場合はそれも通知する       */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/* 備考   : 無し                                                             */
/***************************************************************************/
void hw_can_rec_int( void );

#endif /* HW_CAN_H_ */
```

11.8.4 依存部ソースファイル

【hw_can.c】

```
/***************************************************************************/
/*   インクルードファイル                                                   */
/***************************************************************************/
#include "hw_can.h"

/***************************************************************************/
/*   定数定義                                                               */
/***************************************************************************/
```

```
/******************************************************************************/
/*   変数定義                                                                   */
/******************************************************************************/
/* メッセージスロットスロットレジスタテーブル */
UINT8 * const st_CanMsgSltTbl[16] = {
        {
                &COMCTL0,  &COMCTL1,  &COMCTL2,  &COMCTL3,
                &COMCTL4,  &COMCTL5,  &COMCTL6,  &COMCTL7,
                &COMCTL8,  &COMCTL9,  &COMCTL10, &COMCTL11,
                &COMCTL12, &COMCTL13, &COMCTL14, &COMCTL15,

        }
};

/******************************************************************************/
/*   外部宣言                                                                   */
/******************************************************************************/


/******************************************************************************/
/* 関数名 : void hw_can_init( void )                                            */
/* 役割   : CAN0スロットの初期化                                                  */
/* 戻り値 : 無し                                                                 */
/* 引数   : 無し                                                                 */
/* 備考   : 無し                                                                 */
/******************************************************************************/
void hw_can_init( void )
{
        /* CANスリープモード解除 */
        SLEEP_C0SLPR = CAN_HIGH;

        /* CANリセットモード */
        C0CTLR0 |= CAN_RESET_MODE;

        /* リセットフラグが1になるまで待つ */
        while( CAN_HIGH != STATE_RESET_C0STR );

        /* CANモードレジスタ 通常動作モードに設定 */
        C0MDR = CAN_RUN_MODE;

        /* CAN制御レジスタ設定 */
        LOOPBACK_C0CTLR0 = CAN_LOOPBACK;
        BASICCAN_C0CTLR0 = CAN_BASICCAN;
        TSPRE0_C0CTLR0   = CAN_TSPRE0;
        TSPRE1_C0CTLR0   = CAN_TSPRE1;
        INTSEL_C0CTLR1   = CAN_INTSEL;

        /* CANボーレートプリスケーラ設定 */
        C0BRP = CAN_BRP;

        /* CANコンフィグレーションレジスタ設定 */
        SAM_C0CONR   = CAN_SAM;
        PTS0_C0CONR  = CAN_PTS0;
        PTS1_C0CONR  = CAN_PTS1;
        PTS2_C0CONR  = CAN_PTS2;
        PBS10_C0CONR = CAN_PBS10;
        PBS11_C0CONR = CAN_PBS11;
        PBS12_C0CONR = CAN_PBS12;
        PBS20_C0CONR = CAN_PBS20;
        PBS21_C0CONR = CAN_PBS21;
        PBS22_C0CONR = CAN_PBS22;
        SJW0_C0CONR  = CAN_SJW0;
```

```
        SJW1_COCONR  = CAN_SJW1;

        /* CAN拡張IDレジスタ設定 */
        COIDR = CAN_SET_ID;

        /* CANスロット割り込みマスクレジスタ設定 */
        COSIMKR = CAN_SLOT_INT;

        /* CANエラー割り込みマスク設定 */
        BOIM_COEIMKR = CAN_BUSOFF_INT;
        EPIM_COEIMKR = CAN_ERRPAS_INT;
        BEIM_COEIMKR = CAN_BUSERR_INT;

        /* 割り込み制御レジスタ設定 */
        ILVL0_CAN0IC = CAN_HIGH;
        ILVL1_CAN0IC = CAN_HIGH;
        ILVL2_CAN0IC = CAN_HIGH;

        /* 割り込み許可レジスタ設定 */
        CAN0OE = CAN_HIGH;                              /* CAN0受信割り込み許可 */

        /* マスクレジスタ設定モードに切り替え */
        BANKSEL_COCTLR1 = CAN_HIGH;

        /* CANグローバルマスクレジスタ設定 */
        COGMR0    = (UINT8)(( CAN_GM_SID >> 6 ) & 0x1F );
        COGMR1    = (UINT8)( CAN_GM_SID & 0x3F );

        /* CANローカルマスクレジスタA設定 */
        COLMAR0   = (UINT8)(( CAN_LMA_SID >> 6 ) & 0x1F );
        COLMAR1   = (UINT8)( CAN_LMA_SID & 0x3F );

        /* CANローカルマスクレジスタB設定 */
        COLMBR0   = (UINT8)(( CAN_LMB_SID >> 6 ) & 0x1F );
        COLMBR1   = (UINT8)( CAN_LMB_SID & 0x3F);

        /* メッセージスロット制御レジスタ、              */
        /* シングルショットレジスタ設定モードに切り替え */
        BANKSEL_COCTLR1 = CAN_LOW;

        /* CANシングルショット制御レジスタ設定 */
        COSSCTLR = CAN_SINGLE_S;              /* シングルショット使用しない */

        COCTLR0 &= ~CAN_RESET_MODE;           /* CANリセットモード解除 */

        /* リセットフラグが0になるまで待つ */
        while( CAN_LOW != STATE_RESET_COSTR );

        /* P7_6をCAN0出力ポートに設定 */
        P7_6   = 1;
        PD7_6  = 1;
        PSC_6  = 1;
        PSL1_6 = 0;
        PS1_6  = 1;

        /* P7_7をCAN0入力ポートに設定 */
        PD7_7 = 0;
        PS1_7 = 0;
        IPS3  = 0;
}

/******************************************************************************/
/* 関数名 : UINT8 hw_can_set_trm( UINT16 id, UINT8 dlc, UINT8 *data )        */
/* 役割   : メッセージスロット0を通して、メッセージスロット0に               */
```

```
/*          送信ID・送信データ長・送信データを設定し、送信要求を出す。       */
/*          送信データは送信データ長で指定されたデータ数だけ設定する。       */
/* 戻り値 : UINT8 ret                                                        */
/*            CAN_E_OK      - 正常終了                                       */
/*            CAN_E_RUNNING - 送信中                                         */
/* 引数   : UINT16 id       - 送信する標準ID ( 0x0000 - 0x07FF)              */
/*          UINT8  dlc      - 送信データ長 ( 0 - 8 )                         */
/*          UINT8  *data    - 送信データを格納した配列の先頭ポインタ         */
/* 備考   : 無し                                                             */
/****************************************************************************/
UINT8 hw_can_set_trm( UINT16 id, UINT8 dlc, UINT8 *data )
{
        UINT8    ret = CAN_E_OK;    /* 戻り値初期化 */
        UINT8    cnt = 0;

        /* 送信処理中か確認 */
        if( CAN_CLEAR_8B != ( *st_CanMsgSltTbl[0] & CAN_SET_TRAMACTIVE )) {
                ret = CAN_E_RUNNING;
        }
        else {
                /* メッセージスロット0制御レジスタクリア */
                *st_CanMsgSltTbl[0] = CAN_CLEAR_8B;

                /* スロットバッファ0とメッセージスロット0を繋ぐ */
                C0SBS = ( CAN_CLEAR_8B | ( C0SBS & 0xf0 ));

                /* スロットバッファ0 標準ID設定 */
                SIDH_C0SLOT0 = (UINT8)(( id >> 6 ) & 0x1F );
                SIDL_C0SLOT0 = (UINT8)( id & 0x3F );

                /* スロットバッファ0 データ長設定 */
                DLC_C0SLOT0 = dlc;

                /* スロットバッファ0 送信データ格納 */
                for( cnt = 0; cnt < dlc; cnt++ ) {
                        c0slot0_addr.mnb.data[cnt] = *data;
                        data++;
                }

                /* 送信要求 */
                *st_CanMsgSltTbl[0] = CAN_TRM_REQ;
        }

        return ret;
}


/****************************************************************************/
/* 関数名 : UINT8 hw_can_set_rec( UINT16 id, UINT8 slot )                    */
/* 役割   : 指定されたメッセージスロットをスロットバッファ1を割り当てる      */
/*          スロットバッファ1を通して、メッセージスロットに受信IDを設定し、  */
/*          受信要求を出す                                                   */
/* 戻り値 : UINT8 ret                                                        */
/*            CAN_E_OK      - 正常終了                                       */
/*            CAN_E_RUNNING - 受信中                                         */
/* 引数   : UINT16 id       - 受信設定する標準ID ( 0x0000 - 0x07FF )         */
/*          UINT8  slot     - 受信設定するメッセージスロット ( 1 - 15 )      */
/* 備考   : 無し                                                             */
/****************************************************************************/
UINT8 hw_can_set_rec( UINT16 id , UINT8 slot )
{
        UINT8    ret = CAN_E_OK;    /* 戻り値初期化 */

        /* 指定されたメッセージスロットが受信処理中か確認 */
```

```
	if( CAN_CLEAR_8B != ( *st_CanMsgSltTbl[slot] & CAN_SET_INVALDATA )) {
		ret = CAN_E_RUNNING;
	}
	else {
		/* 指定されたメッセージスロット制御レジスタクリア */
		*st_CanMsgSltTbl[slot] = CAN_CLEAR_8B;


		/* スロットバッファ1と繋ぐメッセージスロットを設定 */
		C0SBS = (( slot << 4 ) | ( C0SBS & 0x0f ));


		/* スロットバッファ1 標準ID設定 */
		SIDH_C0SLOT1 = (UINT8)(( id >> 6 ) & 0x1F );
		SIDL_C0SLOT1 = (UINT8)( id & 0x3F );

		/* 受信要求 */
		*st_CanMsgSltTbl[slot] = CAN_REC_REQ;
	}

	return ret;
}


/****************************************************************************/
/* 関数名 : void hw_can_rec_int( void )                                    */
/* 役割   : メッセージスロット1の受信終了割り込みで起こされる              */
/*          データ長とデータを取得し、コールバックルーチンで上位層に渡す   */
/*          データ取得中にオーバランエラーが起きた場合はそれも通知する     */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/* 備考   : 無し                                                            */
/****************************************************************************/
void hw_can_rec_int( void )
{
	UINT8 cnt  = 0;
	UINT8 slot = 0;

	/* コールバック関数の引数用変数 */
	UINT8  rec_err      = CAN_E_OK;
	UINT16 id_buff      = 0;
	UINT8  dlc_buff     = 0;
	UINT8  data_buff[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };


	/* 受信完了したメッセージスロット番号を保持 */
	slot = C0STRL;

	/* 受信完了したメッセージスロット番号をメッセージバッファ1と繋ぐ */
	C0SBS = (( slot << 4 ) | ( C0SBS & 0x0f ));

	/* NEWDATAビットをLOWにする */
	*st_CanMsgSltTbl[slot] &= CAN_CLEAR_NEWDATA;

	/* スロットバッファ1 標準ID設定 */
	id_buff = ((( SIDH_C0SLOT1 & 0x1F ) << 6 ) | ( SIDL_C0SLOT1 & 0x3F ));

	/* 受信データ長保持 */
	dlc_buff = DLC_C0SLOT1;

	/* 受信データ保持 */
	for( cnt = 0; cnt < dlc_buff; cnt++ ) {
		data_buff[cnt] = c0slot1_addr.mnb.data[cnt];
	}
```

```
	/* 受信データ保持中に新たなデータを上書きしていないかチェック */
	if( CAN_CLEAR_8B != ( *st_CanMsgSltTbl[slot] & CAN_SET_NEWDATA )) {
		/* オーバランエラーを通知する */
		rec_err = CAN_E_OVERRUN;
		/* 次の受信が出来るようにする */
		*st_CanMsgSltTbl[slot] &= CAN_CLEAR_NEWDATA;
	}
	else {
		/* 何もしない */
	}

	/* コールバックルーチンを呼び出し */
	CanRecCbr( rec_err, id_buff, dlc_buff, data_buff );

	/* 割り込み要求をクリア */
	COSISTR = CAN_CLEAR_16B;
	CANOOR = CAN_LOW;
}
```

11.9 9.3 章 - CAN デバイスドライバを用いたアプリケーションの作成

11.9.1 （送信アプリ）OIL ファイル

【sample.oil】

```
/*
 *  定義部のインクルード
 */
#include "implementation.oil"


/*
 *  実装部
 */
CPU current {

    /* syslibのシリアル機能を使用する */
    #include <serial.oil>

    /* 100μsecシステムタイマ機能を使用する */
    #include <t100us_timer.oil>

    /*
     *  OS定義
     */
    OS os {
        STATUS = STANDARD;
        STARTUPHOOK = TRUE;
        ERRORHOOK = FALSE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        USEGETSERVICEID = TRUE;
        USEPARAMETERACCESS = TRUE;
        USERESSCHEDULER = TRUE;
    };


    /*
     *  アプリケーションモード定義
     */
    APPMODE AppMode1 {};


    /*
     *  タスク定義
     */
    /* CANタスク    */
    TASK CanTask {
        AUTOSTART = TRUE {
            APPMODE = AppMode1;
        };
        PRIORITY = 8;
        STACKSIZE = 0x0180;
        ACTIVATION = 1;
        SCHEDULE = FULL;
        EVENT = CanTrmEvt;
    };
```

```
    /* スイッチタスク    */
    TASK SwitchTask {
        AUTOSTART = TRUE {
            APPMODE = AppMode1;
        };
        PRIORITY = 7;
        STACKSIZE = 0x0180;
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };


    /*
     *  割込みサービスルーチン定義
     */
    /* CAN受信割込み処理 */
    ISR hw_can_rec_int {
        CATEGORY = 1;
        ENTRY = 53;
        PRIORITY = 6;
    };


    /*
     *  イベント定義
     */
    /* CAN送信イベント */
    EVENT CanTrmEvt {
        MASK = AUTO;
    };


    /*
     *  アラーム定義
     */
    /* Canタスク周期アラーム    */
    ALARM CanCycArm {
        COUNTER = T100usTimerCnt;
        ACTION  = ACTIVATETASK {
            TASK = CanTask;
        };
        AUTOSTART = TRUE {
            APPMODE = AppMode1;
            ALARMTIME = 1;
            CYCLETIME = 50;
        };
    };

    /* Switchタスク周期アラーム    */
    ALARM SwitchCycArm {
        COUNTER = T100usTimerCnt;
        ACTION  = ACTIVATETASK {
            TASK = SwitchTask;
        };
        AUTOSTART = TRUE {
            APPMODE = AppMode1;
            ALARMTIME = 1;
            CYCLETIME = 30;
        };
    };


};
```

11.9.2　（送信アプリ）ヘッダファイル

【sample.h】

```
/******************************************************************************/
/*   インクルードファイル                                                      */
/******************************************************************************/

/******************************************************************************/
/*   定数定義                                                                  */
/******************************************************************************/
#define CAN_TRM_FLAG_CLEAR    ((UINT8)9)    /* CAN送信フラグクリア用マクロ */
#define CAN_TRM_FLAG_0        ((UINT8)0)    /* CAN送信フラグ用マクロ0 */
#define CAN_TRM_FLAG_1        ((UINT8)1)    /* CAN送信フラグ用マクロ1 */
#define CAN_TRM_FLAG_2        ((UINT8)2)    /* CAN送信フラグ用マクロ2 */
#define CAN_TRM_FLAG_3        ((UINT8)3)    /* CAN送信フラグ用マクロ3 */

/******************************************************************************/
/*   変数定義                                                                  */
/******************************************************************************/

/******************************************************************************/
/*   外部宣言                                                                  */
/******************************************************************************/
```

11.9.3　（送信アプリ）ソースファイル

【sample.c】

```
/************************************************************************/
/*                                                                      */
/*  FILE        :sample.c                                               */
/*  DATE        :Fri, Oct 03, 2008                                      */
/*  DESCRIPTION :main program file.                                     */
/*  CPU GROUP   :85(ROM512K)                                            */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.4.8).     */
/*                                                                      */
/************************************************************************/

/******************************************************************************/
/*   インクルードファイル                                                      */
/******************************************************************************/
#include "kernel.h"
#include "t100us_timer.h"
#include "serial.h"
#include "kernel_id.h"
#include "sample.h"

#include "can.h"
#include "sw.h"

/******************************************************************************/
/*   定数定義                                                                  */
/******************************************************************************/

/******************************************************************************/
/*   変数定義                                                                  */
/******************************************************************************/
```

```
/* CAN送信フラグ */
static UINT8 can_trm_flag = CAN_TRM_FLAG_CLEAR;
/* CAN送信IDテーブル */
static const UINT16 can_trm_id[4]   = { 0x0001, 0x0002, 0x0003, 0x0004 };
/* CAN送信データ長テーブル */
static const UINT8  can_trm_dlc[4]  = { 1, 1, 1, 1 };
/* CAN送信データテーブル */
static const UINT8  can_trm_data[4] = { 0x11, 0x22, 0x33, 0x44 };

/******************************************************************************/
/*   外部宣言                                                                  */
/******************************************************************************/

/******************************************************************************/
/*   内部宣言                                                                  */
/******************************************************************************/
void main( void );

DeclareTask( CanTask );
DeclareTask( SwitchTask );
DeclareEvent( CanTrmEvt );

TASK( CanTask );
TASK( SwitchTask );


/******************************************************************************/
/* 関数名 : void main( void )                                                  */
/* 役割   : カーネルを起動する                                                  */
/* 戻り値 : 無し                                                               */
/* 引数   : 無し                                                               */
/* 備考   : 無し                                                               */
/******************************************************************************/
void main( void )
{
        /*
         *  カーネル起動
         */
        StartOS( AppMode1 );

}


/******************************************************************************/
/* 関数名 : TASK( CanTask )                                                    */
/* 役割   : イベントが発生した時、送信フラグを参照してCAN送信を行う             */
/*          その後、送信フラグをクリアし、イベント待ちに戻る                    */
/* 戻り値 : 無し                                                               */
/* 引数   : 無し                                                               */
/* 備考   : 無し                                                               */
/******************************************************************************/
TASK( CanTask )
{
        while( 1 ) {
                /* イベント待ち */
                WaitEvent( CanTrmEvt );

                /* イベントクリア */
                ClearEvent( CanTrmEvt );

                /* 送信フラグを用いてCAN送信 */
                CanSetTrm( can_trm_id[can_trm_flag],
                           can_trm_dlc[can_trm_flag],
                           &can_trm_data[can_trm_flag] );
```

```
			/* 送信フラグクリア */
			can_trm_flag = CAN_TRM_FLAG_CLEAR;
		}

		/* タスク終了 */
		TerminateTask();
}


/****************************************************************************/
/* 関数名 : TASK( SwitchTask )                                              */
/* 役割   : スイッチの状態を取得、押下されていた場合、                      */
/*          スイッチに対応したCAN送信フラグの更新とイベント発生処理を行う   */
/*          また、複数のスイッチが同時に押下された場合、                    */
/*          処理優先度はスイッチ3 > 4 > 5 > 6 とする                        */
/* 戻り値 : 無し                                                            */
/* 引数   : 無し                                                            */
/* 備考   : 本タスクは30msec周期で起動される                                */
/****************************************************************************/
TASK( SwitchTask )
{
		UINT8 sw_state[4];

		/* 各スイッチの状態を取得 */
		SwGetPush3( &sw_state[0] );
		SwGetPush4( &sw_state[1] );
		SwGetPush5( &sw_state[2] );
		SwGetPush6( &sw_state[3] );

		/* スイッチ押下時に対応したCAN送信フラグの更新とイベント発生処理を行う */
		/* スイッチ3→CAN送信フラグ0 */
		if( SW_ON == sw_state[0] ) {
			can_trm_flag = CAN_TRM_FLAG_0;
			SetEvent( CanTask, CanTrmEvt );
		}
		/* スイッチ4→CAN送信フラグ1 */
		else if( SW_ON == sw_state[1] ) {
			can_trm_flag = CAN_TRM_FLAG_1;
			SetEvent( CanTask, CanTrmEvt );
		}
		/* スイッチ5→CAN送信フラグ2 */
		else if( SW_ON == sw_state[2] ) {
			can_trm_flag = CAN_TRM_FLAG_2;
			SetEvent( CanTask, CanTrmEvt );
		}
		/* スイッチ6→CAN送信フラグ3 */
		else if( SW_ON == sw_state[3] ) {
			can_trm_flag = CAN_TRM_FLAG_3;
			SetEvent( CanTask, CanTrmEvt );
		}
		else {
			/* 何もしない */
		}

		/* タスク終了 */
		TerminateTask();
}


/****************************************************************************/
/* 関数名 : void CanRecCbr( UINT8 rec_err, UINT16 id UINT8 dlc, UINT8 *data )*/
/* 役割   : CAN受信割り込みから呼び出される                                 */
/*          CAN受信完了を通知するコールバック関数                           */
```

```
/*          本アプリではCAN受信を行わないため、何も処理はしない              */
/* 戻り値 : 無し                                                          */
/* 引数   : UINT8  rec_err - 受信成否通知( CAN_E_OK, CAN_E_OVERRUN_E )     */
/*          UINT16 id      - 受信ID( 0x0000 - 0x07FF )                    */
/*          UINT8  dlc     - 受信データ長 ( 0 - 8 )                        */
/*          UINT8  *data   - 受信データ格納した配列の先頭アドレス           */
/* 備考   : CAN受信割り込み依存部関数hw_can_rec_intによって呼び出される     */
/****************************************************************************/
void CanRecCbr( UINT8 rec_err, UINT16 id, UINT8 dlc, UINT8 *data )
{

}



/****************************************************************************/
/* 関数名 : void StartupHook ( void )                                     */
/* 役割   : 各デバイスドライバの初期化                                     */
/* 戻り値 : 無し                                                          */
/* 引数   : 無し                                                          */
/* 備考   : 無し                                                          */
/****************************************************************************/
#ifdef USE_STARTUPHOOK
void StartupHook( void )
{
        /* システムタイマ初期化 */
        InitT100usTimer();

        /* スイッチデバイスドライバ初期化 */
        SwInit();

        /* CANデバイスドライバ初期化 */
        CanInit();

}       /* StartupHook     */
#endif /* USE_STARTUPHOOK */
```

11.9.4 （受信アプリ）OIL ファイル

【sample.oil】

```
/*
 *  定義部のインクルード
 */
#include "implementation.oil"


/*
 *  実装部
 */
CPU current {

/* syslibのシリアル機能を使用する */
#include <serial.oil>

/* 100μsecシステムタイマ機能を使用する */
#include <t100us_timer.oil>

    /*
     *  OS定義
     */
    OS os {
```

```
        STATUS = STANDARD;
        STARTUPHOOK = TRUE;
        ERRORHOOK = FALSE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        USEGETSERVICEID = TRUE;
        USEPARAMETERACCESS = TRUE;
        USERESSCHEDULER = TRUE;
    };

    /*
     *  アプリケーションモード定義
     */
    APPMODE AppMode1 {};

    /*
     *  タスク定義
     */
    /* スイッチタスク    */
    TASK DispTask {
        AUTOSTART = TRUE {
            APPMODE = AppMode1;
        };
        PRIORITY = 7;
        STACKSIZE = 0x0180;
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };


    /*
     *  割込みサービスルーチン定義
     */
    /* CAN受信割込み処理 */
    ISR hw_can_rec_int {
        CATEGORY = 1;
        ENTRY = 53;
        PRIORITY = 6;
    };

    /*
     *  イベント定義
     */


    /*
     *  アラーム定義
     */
    /* Dispタスク周期アラーム    */
    ALARM DispCycArm {
        COUNTER = T100usTimerCnt;
        ACTION  = ACTIVATETASK {
            TASK = DispTask;
        };
        AUTOSTART = TRUE {
            APPMODE = AppMode1;
            ALARMTIME = 1;
            CYCLETIME = 1000;
        };
    };


};
```

11.9.5　（受信アプリ）ヘッダファイル

【sample.h】

```
/****************************************************************************/
/*   インクルードファイル                                                    */
/****************************************************************************/

/****************************************************************************/
/*   定数定義                                                                */
/****************************************************************************/
/* CAN受信ID変数用初期値 */
#define CAN_REC_ID_INIT        ((UINT16)0xffff)

/* CAN受信ID設定値 */
#define CAN_REC_ID_SLOT1    ((UINT16)0x0001)
#define CAN_REC_ID_SLOT2    ((UINT16)0x0002)
#define CAN_REC_ID_SLOT3    ((UINT16)0x0003)
#define CAN_REC_ID_SLOT4    ((UINT16)0x0004)

/****************************************************************************/
/*   変数定義                                                                */
/****************************************************************************/

/****************************************************************************/
/*   外部宣言                                                                */
/****************************************************************************/
```

11.9.6　（受信アプリ）ソースファイル

【sample.c】

```
/**********************************************************************/
/*                                                                    */
/*  FILE        :sample.c                                             */
/*  DATE        :Fri, Oct 03, 2008                                    */
/*  DESCRIPTION :main program file.                                   */
/*  CPU GROUP   :85(ROM512K)                                          */
/*                                                                    */
/*  This file is generated by Renesas Project Generator (Ver.4.8).    */
/*                                                                    */
/**********************************************************************/

/****************************************************************************/
/*   インクルードファイル                                                    */
/****************************************************************************/
#include "kernel.h"
#include "t100us_timer.h"
#include "serial.h"
#include "kernel_id.h"
#include "sample.h"

#include "can.h"
#include "lcd.h"

/****************************************************************************/
/*   定数定義                                                                */
```

```
/*****************************************************************************/

/*****************************************************************************/
/*   変数定義                                                                */
/*****************************************************************************/
/* CAN受信ID */
static UINT16 can_rec_id = CAN_REC_ID_INIT;
/* CAN送信データ */
static UINT8  can_rec_data = 0x00;
/* LCD表示文字列 */
static UINT8 lcd_str[LCD_DEV_LINE][LCD_DEV_DIGIT] = { "ID  /", "DATA/" };

/*****************************************************************************/
/*   外部宣言                                                                */
/*****************************************************************************/

/*****************************************************************************/
/*   内部宣言                                                                */
/*****************************************************************************/
void main( void );

DeclareTask(DispTask);

TASK( DispTask );


/*****************************************************************************/
/* 関数名 : void main( void )                                                */
/* 役割   : カーネルを起動する                                               */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/* 備考   : 無し                                                             */
/*****************************************************************************/
void main( void )
{
        /*
         *  カーネル起動
         */
        StartOS( AppMode1 );
}


/*****************************************************************************/
/* 関数名 : TASK( DispTask )                                                 */
/* 役割   : CAN受信IDが初期状態で無い場合、                                  */
/*          受信IDと受信データを文字列に変換、LCD表示文字列に格納し、        */
/*          LCDに表示する                                                    */
/* 戻り値 : 無し                                                             */
/* 引数   : 無し                                                             */
/* 備考   : 本タスクは100msec周期で起動される                                */
/*****************************************************************************/
TASK( DispTask )
{
        /* CAN受信IDが初期状態で無いか確認 */
        if( CAN_REC_ID_INIT != can_rec_id ) {
                /* 数値を文字列に変換・格納 */
                ConvShort2HexStr( &lcd_str[0][5], can_rec_id );
                ConvByte2HexStr( &lcd_str[1][5], can_rec_data );

                /* LCD表示 */
                LcdWriteLine( 0, lcd_str[0], 1 );
                LcdWriteLine( 0, lcd_str[1], 2 );
        }
        else {
```

```
			/* 何もしない */
		}

		/* タスク終了 */
		TerminateTask();
}


/***************************************************************************/
/* 関数名 : void CanRecCbr( UINT8 rec_err, UINT16 id UINT8 dlc, UINT8 *data )*/
/* 役割   : CAN受信割り込みから呼び出される                                    */
/*          CAN受信完了を通知するコールバック関数                              */
/*          引数の正常受信orオーバランエラー発生、受信完了したID、             */
/*          受信データ長、受信データを格納した配列の先頭アドレスを用いて       */
/*          受信IDと受信データを変数に格納する                                 */
/* 戻り値 : 無し                                                               */
/* 引数   : UINT8  rec_err - 受信成否通知( CAN_E_OK, CAN_E_OVERRUN_E )       */
/*          UINT16 id      - 受信ID( 0x0000 - 0x07FF )                        */
/*          UINT8  dlc     - 受信データ長 ( 0 - 8 )                           */
/*          UINT8  *data   - 受信データ格納した配列の先頭アドレス             */
/* 備考   : CAN受信割り込み依存部関数hw_can_rec_intによって呼び出される      */
/***************************************************************************/
void CanRecCbr( UINT8 rec_err, UINT16 id, UINT8 dlc, UINT8 *data )
{
		/* 受信エラーが無い場合、受信IDと受信データを格納 */
		if( CAN_E_OK == rec_err ) {
			can_rec_id   = id;
			can_rec_data = data[0];
		}
		else {
			/* 何もしない */
		}
}


/***************************************************************************/
/* 関数名 : void StartupHook ( void )                                        */
/* 役割   : 各デバイスドライバの初期化、LCDの初期表示、CANの受信設定        */
/* 戻り値 : 無し                                                               */
/* 引数   : 無し                                                               */
/* 備考   : 無し                                                               */
/***************************************************************************/
#ifdef USE_STARTUPHOOK
void StartupHook( void )
{
		/* システムタイマ初期化 */
		InitT100usTimer();

		/* LCDデバイスドライバ初期化 */
		LcdInit();

		/* CANデバイスドライバ初期化 */
		CanInit();

		/* LCDクリア */
		LcdCtlDisplay( 0, LCD_CTL_CLRDISPLAY );

		/* LED初期表示 */
		LcdWriteLine( 0, lcd_str[0], 1 );
		LcdWriteLine( 0, lcd_str[1], 2 );

		/* CAN受信設定 */
		CanSetRec( CAN_REC_ID_SLOT1, 1 );
		CanSetRec( CAN_REC_ID_SLOT2, 2 );
```

```
          CanSetRec( CAN_REC_ID_SLOT3, 3 );
          CanSetRec( CAN_REC_ID_SLOT4, 4 );

}         /* StartupHook      */
#endif /* USE_STARTUPHOOK */
```

12 Appendix B（MISRA-C ルール一覧）

以下に、MISRA-C:2004 のルール一覧を記載する。尚、「組込み開発者におくる MISRA-C2004 C 言語利用の高信頼化ガイド」（MISRA-C 研究会著）を参照とした。

カテゴリ	No	必要/推奨	ルール内容
環境	ルール 1.1	(必要)	ISO/IEC 9899:1990
	ルール 1.2	(必要)	未定義・未規定動作
	ルール 1.3	(必要)	複数のコンパイラや言語の使用
	ルール 1.4	(必要)	外部識別氏名
	ルール 1.5	(推奨)	浮動小数点規格
言語拡張	ルール 2.1	(必要)	アセンブリ言語
	ルール 2.2	(必要)	コメント
	ルール 2.3	(必要)	コメント内の/*
	ルール 2.4	(推奨)	コメントアウト
文書化	ルール 3.1	(必要)	処理系定義の動作
	ルール 3.2	(必要)	文字集合及びエンコーディング
	ルール 3.3	(推奨)	整数除算
	ルール 3.4	(必要)	#pragma 指令
	ルール 3.5	(必要)	ビットフィールド
	ルール 3.6	(必要)	ライブラリ
文字集合	ルール 4.1	(必要)	拡張表記
	ルール 4.2	(必要)	3 文字表記
識別子	ルール 5.1	(必要)	31 文字を超える識別子
	ルール 5.2	(必要)	識別子の隠ぺい
	ルール 5.3	(必要)	typedef 名
	ルール 5.4	(必要)	タグ名
	ルール 5.5	(推奨)	オブジェクト・関数識別子の再使用
	ルール 5.6	(推奨)	ネームスペース
	ルール 5.7	(推奨)	識別子の再使用
型	ルール 6.1	(必要)	char 型
	ルール 6.2	(必要)	signed char 型・unsigned char 型
	ルール 6.3	(推奨)	基本型の代わりの typedef
	ルール 6.4	(必要)	ビットフィールド
	ルール 6.5	(必要)	signed int 型のビットフィールド
定数	ルール 7.1	(必要)	(0 以外の)8 進定数
宣言及び定義	ルール 8.1	(必要)	プロトタイプ宣言
	ルール 8.2	(必要)	オブジェクト・関数の宣言・定義
	ルール 8.3	(必要)	仮引数の型・戻り値の型
	ルール 8.4	(必要)	オブジェクト・関数の複数回宣言
	ルール 8.5	(必要)	ヘッダファイル内の定義
	ルール 8.6	(必要)	関数の宣言
	ルール 8.7	(必要)	オブジェクトの定義
	ルール 8.8	(必要)	外部オブジェクト・外部関数
	ルール 8.9	(必要)	外部結合の識別子

<table>
<tr><td rowspan="3"></td><td>ルール 8.10</td><td>(必要)</td><td>ファイルスコープのオブジェクト・関数</td></tr>
<tr><td>ルール 8.11</td><td>(必要)</td><td>static 記憶域クラスの指定子</td></tr>
<tr><td>ルール 8.12</td><td>(必要)</td><td>外部結合をもつ配列</td></tr>
<tr><td rowspan="3">初期化</td><td>ルール 9.1</td><td>(必要)</td><td>自動変数</td></tr>
<tr><td>ルール 9.2</td><td>(必要)</td><td>配列・構造体の初期化</td></tr>
<tr><td>ルール 9.3</td><td>(必要)</td><td>列挙子リスト</td></tr>
<tr><td rowspan="6">算術型変換</td><td>ルール 10.1</td><td>(必要)</td><td>整数型の暗黙的変換</td></tr>
<tr><td>ルール 10.2</td><td>(必要)</td><td>浮動小数点型の暗黙的変換</td></tr>
<tr><td>ルール 10.3</td><td>(必要)</td><td>整数型のキャスト</td></tr>
<tr><td>ルール 10.4</td><td>(必要)</td><td>浮動小数点型のキャスト</td></tr>
<tr><td>ルール 10.5</td><td>(必要)</td><td>ビット単位の演算子~と<<</td></tr>
<tr><td>ルール 10.6</td><td>(必要)</td><td>符号なし型の定数</td></tr>
<tr><td rowspan="5">ポインタ型の変換</td><td>ルール 11.1</td><td>(必要)</td><td>関数ポインタ</td></tr>
<tr><td>ルール 11.2</td><td>(必要)</td><td>オブジェクトポインタ</td></tr>
<tr><td>ルール 11.3</td><td>(推奨)</td><td>ポインタ型と汎整数型とのキャスト</td></tr>
<tr><td>ルール 11.4</td><td>(推奨)</td><td>オブジェクト型を指すポインタのキャスト</td></tr>
<tr><td>ルール 11.5</td><td>(必要)</td><td>const 修飾・volatile 修飾</td></tr>
<tr><td rowspan="13">式</td><td>ルール 12.1</td><td>(推奨)</td><td>演算子優先順位への依存</td></tr>
<tr><td>ルール 12.2</td><td>(必要)</td><td>式の評価</td></tr>
<tr><td>ルール 12.3</td><td>(必要)</td><td>sizeof 演算子</td></tr>
<tr><td>ルール 12.4</td><td>(必要)</td><td>論理演算子&&・||の右側オペランド</td></tr>
<tr><td>ルール 12.5</td><td>(必要)</td><td>論理演算子&&・||のオペランド</td></tr>
<tr><td>ルール 12.6</td><td>(推奨)</td><td>論理演算子(&&・|・!)|のオペランド</td></tr>
<tr><td>ルール 12.7</td><td>(必要)</td><td>ビット単位の演算子</td></tr>
<tr><td>ルール 12.8</td><td>(必要)</td><td>シフト演算子</td></tr>
<tr><td>ルール 12.9</td><td>(必要)</td><td>単項マイナス(−)演算子</td></tr>
<tr><td>ルール 12.10</td><td>(必要)</td><td>カンマ演算子</td></tr>
<tr><td>ルール 12.11</td><td>(推奨)</td><td>符号なし整数定数式の評価</td></tr>
<tr><td>ルール 12.12</td><td>(必要)</td><td>浮動小数点数のビット表現</td></tr>
<tr><td>ルール 12.13</td><td>(推奨)</td><td>インクリメント(++)・デクリメント(−−)</td></tr>
<tr><td rowspan="7">制御文の式</td><td>ルール 13.1</td><td>(必要)</td><td>ブール値が生成される式</td></tr>
<tr><td>ルール 13.2</td><td>(推奨)</td><td>0 との比較テスト</td></tr>
<tr><td>ルール 13.3</td><td>(必要)</td><td>浮動小数点式</td></tr>
<tr><td>ルール 13.4</td><td>(必要)</td><td>for 文の制御式</td></tr>
<tr><td>ルール 13.5</td><td>(必要)</td><td>for 文の 3 つの式</td></tr>
<tr><td>ルール 13.6</td><td>(必要)</td><td>for ループのカウンタ</td></tr>
<tr><td>ルール 13.7</td><td>(必要)</td><td>結果が不変のブール演算</td></tr>
<tr><td rowspan="10">制御フロー</td><td>ルール 14.1</td><td>(必要)</td><td>到達しないコード</td></tr>
<tr><td>ルール 14.2</td><td>(必要)</td><td>空文</td></tr>
<tr><td>ルール 14.3</td><td>(必要)</td><td>空文の表記</td></tr>
<tr><td>ルール 14.4</td><td>(必要)</td><td>goto 文</td></tr>
<tr><td>ルール 14.5</td><td>(必要)</td><td>continue 文</td></tr>
<tr><td>ルール 14.6</td><td>(必要)</td><td>break 文</td></tr>
<tr><td>ルール 14.7</td><td>(必要)</td><td>関数の出口</td></tr>
<tr><td>ルール 14.8</td><td>(必要)</td><td>繰返し文の本体</td></tr>
<tr><td>ルール 14.9</td><td>(必要)</td><td>"if(式)"の後</td></tr>
<tr><td>ルール 14.10</td><td>(必要)</td><td>if...else if の else 節</td></tr>
<tr><td rowspan="4">switch 文</td><td>ルール 15.1</td><td>(必要)</td><td>switch ラベル</td></tr>
<tr><td>ルール 15.2</td><td>(必要)</td><td>空でない switch 節</td></tr>
<tr><td>ルール 15.3</td><td>(必要)</td><td>switch 文の最後の節</td></tr>
<tr><td>ルール 15.4</td><td>(必要)</td><td>switch 式</td></tr>
</table>

	ルール 15.5	（必要）	switch 文と case 節
関数	ルール 16.1	（必要）	可変個引数をもつ関数
	ルール 16.2	（必要）	関数の再起呼出し
	ルール 16.3	（必要）	関数プロトタイプ宣言の仮引数
	ルール 16.4	（必要）	関数宣言と定義の仮引数の識別子
	ルール 16.5	（必要）	仮引数をもたない関数
	ルール 16.6	（必要）	実引数の個数
	ルール 16.7	（推奨）	ポインタ仮引数
	ルール 16.8	（必要）	戻り値の型が非の void の関数
	ルール 16.9	（必要）	関数識別子の使用
	ルール 16.10	（必要）	エラー情報を戻す関数
ポインタ及び配列	ルール 17.1	（必要）	ポインタ算術
	ルール 17.2	（必要）	ポインタ減算
	ルール 17.3	（必要）	ポインタ型の大小比較
	ルール 17.4	（必要）	ポインタ算術で許される形式
	ルール 17.5	（推奨）	ポインタ間接参照
	ルール 17.6	（必要）	自動記憶域のオブジェクト
構造体及び共用体	ルール 18.1	（必要）	構造体･共用体の型
	ルール 18.2	（必要）	重複するオブジェクト
	ルール 18.3	（必要）	メモリ領域の再使用
	ルール 18.4	（必要）	共用体の使用
前処理指令	ルール 19.1	（推奨）	#include 文の前
	ルール 19.2	（推奨）	#include におけるヘッダファイル名
	ルール 19.3	（必要）	<filename>・″filename″
	ルール 19.4	（必要）	C マクロ
	ルール 19.5	（必要）	ブロック内の#define・#undef
	ルール 19.6	（必要）	#undef
	ルール 19.7	（推奨）	関数形式マクロ
	ルール 19.8	（必要）	関数形式マクロの呼出し
	ルール 19.9	（必要）	関数形式マクロの引数
	ルール 19.10	（必要）	関数形式マクロの仮引数
	ルール 19.11	（必要）	前処理指令のマクロ識別子
	ルール 19.12	（必要）	#･##前処理演算子
	ルール 19.13	（推奨）	前処理演算子#･##の使用
	ルール 19.14	（必要）	define 前処理演算子
	ルール 19.15	（必要）	ヘッダファイルの複数取り込み予防
	ルール 19.16	（必要）	前処理指令の前処理除外
	ルール 19.17	（必要）	#else,#elif,#endif 前処理指令
標準ライブラリ	ルール 20.1	（必要）	予約済み識別子・標準ライブラリのマクロ･関数
	ルール 20.2	（必要）	標準ライブラリのマクロ･オブジェクト･関数名
	ルール 20.3	（必要）	ライブラリ関数に渡される値
	ルール 20.4	（必要）	ヒープメモリ割り当て
	ルール 20.5	（必要）	エラー指示子 errno
	ルール 20.6	（必要）	<stddef.h>ライブラリの offsetof マクロ
	ルール 20.7	（必要）	setjmp マクロ・longjmp 関数
	ルール 20.8	（必要）	<signal.h>のシグナル処理機能
	ルール 20.9	（必要）	入出力ライブラリ<stdio.h>

	ルール 20.10	（必要）	<stdlib.h>ライブラリ関数 atof,atoi,atol
	ルール 20.11	（必要）	<stdlib.h>ライブラリ関数 abort,exit,getenv
	ルール 20.12	（必要）	<time.h>ライブラリの時間処理関数
実行時誤り	ルール 21.1	（必要）	実行時誤り

参考文献
参考 URL

TOPPERS プロジェクト　TOPPERS Automotive Kernel 外部仕様書 Ver3.00
TOPPERS プロジェクト　TOPPERS Automotive Kernel SG 取扱説明書 Ver5.00
OSEK/VDX Operating System Ver2.2.3
OSEK/VDX System Generation OIL Ver2.5
ルネサステクノロジ　M32C/85 グループ(M32C/85、M32C/85T) ハードウエアマニュアル
CQ 出版社　TECH I Vol.15 リアルタイム/マルチタスクシステムの徹底研究
CQ 出版社　TECH I Vol.17 リアルタイム OS と　組込み技術の基礎
CQ 出版社　TECH I Vol.19 実践リアルタイム OS 活用技法
CQ 出版社　Design Wave Magazine 2003 年 5 月号別冊付録
CQ 出版社　Design Wave Magazine 2005 年 12 月増刊号
CQ 出版社　Interface 2005 年 4 月号
EE Times Japan　クルマに広がるネットワーク（前編）　電子化の背景と車載 LAN の種類
MISRA-C 研究会　組込み開発者におくる MISRA-C 組込みプログラミングの高信頼化ガイド
MISRA-C 研究会　組込み開発者におくる MISRA-C2004 C 言語利用の高信頼化ガイド
@IT MONOist　車載ネットワーク“CAN の仕組み”教えます
（URL : http://monoist.atmarkit.co.jp/fembedded/index/carele_index.html#carnet）
株式会社ルネサスソリューションズ　TTV（Task Trace View）　タスク動作シミュレータ
（URL：http://www.t-engine.org/ja/wp-content/themes/wp.vicuna/html/ttv/files/ttv.html）
（URL：http://en.wikipedia.org/wiki/OBD-II_PIDs)